Technical narratives
analysis, description and representation in the conservation of software-based art

Ensom, Thomas

Awarding institution:
King's College London

TECHNICAL NARRATIVES: ANALYSIS, DESCRIPTION AND REPRESENTATION IN THE CONSERVATION OF SOFTWARE-BASED ART

Thomas C. Ensom

Thesis Submitted in Partial Fulfilment of the Requirements for the Degree of Doctor of Philosophy in Digital Humanities

Faculty of Arts and Humanities, King's College London

Corrections Revision, March 2019
[Minor redactions of sensitive and copyrighted material for e-thesis publication carried out in September 2019]

80,204 words.

Abstract

The term software-based art has emerged from conservation practice over the past decade to describe artworks for which software forms the primary artistic medium. Such works present new challenges for those engaged in the long-term care of collections of modern and contemporary art. They are often technically complex and may employ many inter-related (and sometimes bespoke) components, embedded in a specific technical environment. As a result, software-based artworks are particularly at risk from processes of loss and obsolescence. While progress has been made toward the development of practical strategies for their preservation, how to effectively document them in a conservation context remains poorly understood.

In this thesis, I describe practice-led research which has sought to address this gap using a constructive research approach. I first develop a conceptual framework through which to better understand the problem space, consisting of two parts: an in-depth examination of the characteristics of software as a medium; and an exploration of the document concept and its meaning in relation to the role of the conservator. Using this conceptual framework to further refine my research aims, I examine three topics in detail, seeking to develop practical solutions for each: the analysis and representation of software structures; the extent to which notions of significance and artwork identity might be formalised as documentation; and how the patterns of change which occur in the life of a software-based artwork might be understood and recorded.

In addressing each of these aims, I draw on insights gained in the in-depth study of a set of software-based artwork case studies from the Tate collection and the synthesis of existing theory from a number of related domains. The outcomes of the research have direct relevance to conservation practice, not as formal templates, but rather as a set of flexible and reusable principles and methods that might be applied individually or in conjunction to effectively document a diversity of software-based artwork types.

Table of Contents

List of Tables

List of Figures

Acknowledgements

I dedicate this thesis to my parents, Meriel and Paul Ensom, who have instilled in me a thirst for knowledge and an interest in all things, and without whom I would never have ended up on the winding road that eventually led me to undertake to this PhD.

Thank you to my supervisors Mark Hedges and Pip Laurenson, whose measured guidance kept me on track on the long road to submission. When I saw this project advertised four years ago, I couldn't believe how closely aligned it was with my research interests. I am so grateful that you devised its foundations and entrusted me with its undertaking.

Special thanks to Patricia Falcão, whose patience and insight was incredibly important in shaping my research and keeping me grounded in the reality of Tate's day-to-day work. Your open mind and enthusiasm for the subject continues to inspire me.

Thank you to the entire Time-based Media Conservation team at Tate, past and present, who welcomed me during my time at Tate Stores. I feel very lucky to have worked within such a great team and have learnt a great deal from the experience.

Thank you to Martina Haidvogl, Mark Hellar and Jill Sterrett, who made my research visit to San Francisco Museum of Modern Art not only possible, but such a stimulating experience.

Thank you to all those who generously agreed to be interviewed during my research—Deena Engel, Ben Fino-Radin, Mark Hellar, Joanna Phillips, Klaus Rechert, Eric Kaltman, Jon Ippolito and Gaby Wijers—with extra thanks to Deena and Mark, who first coined the term "technical narrative" that forms a crucial part of title of this thesis. Thanks also to Annet Dekker for her advice in the early phases of this research, and to the PERICLES research team for another set of formative experiences.

Last but by no means least, thank you to my wonderful partner Kitty Clark for her patience during the long hours at my desk and her support during those moments when it started to feel too much. I'm not sure I could have finished it without you, and I'm so glad you were there to enrich my work and enthuse me with your own passion for art.

CHAPTER 1

INTRODUCTION

1.1. Thesis Outline

The term software-based art has emerged from art conservation practice over the past decade to describe a group of artworks for which software forms the primary artistic medium. The characteristics of these works pose new challenges for conservators engaged in the long-term care of collections of modern and contemporary art. They are often technically complex and may employ many inter-related components embedded in a highly specific technical environment. These components often include bespoke code used to achieve particular behaviours or qualities, the underlying complexity of which is typically not apparent from the tangible elements of the work nor from the software's compiled form. As the external technical environment changes through time, it may become increasingly difficult to realise these works, as hardware components become harder to replace and the software platforms employed move towards obsolescence. Software-based artworks can therefore be considered at risk of loss if not properly cared for. While progress has been made toward the development of practical strategies for preserving software-based artworks, how to effectively document them in a conservation context remains poorly understood. In this thesis I aim to address this gap through a practice-led study of the issues involved, and the use of existing theory from a number of related

domains to develop pragmatic approaches to documentation.

I begin by developing a conceptual framework, consisting of two fundamental research strands. This first is the development of a more complete understanding of the characteristics of software as an artistic medium—particularly in relation to the technical characteristics of software and the medium-specific conservation considerations demanded in its treatment and care. The second is the theoretical re-consideration of the delimitation of the concept of *document* and how this relates to the practical undertaking of documentation as a core conservation activity with a variety of purposes—undertaken by both human and machine agents. Taken together, these two research strands form a conceptual framework which allows the identification of three key challenges in the documentation of software-based art, which I address in turn in the subsequent chapters. The first concerns the analysis and representation of the software structures, which form the basis of the software performance that occurs when a work is realised. The second concerns the extent to which notions of significance and artwork identity might be pragmatically formalised as documentation. The third concerns how the patterns of change which occur in the life of a software-based artwork might be understood and recorded. The outcomes of these chapters are not formal templates, but rather offer flexible and reusable principles and methods that might be applied individually or in conjunction to effectively document the great variety of software-based artworks.

This research is intrinsically interdisciplinary in nature and necessitates a novel synthesis of knowledge from digital preservation, art conservation, software engineering and other related domains. While based primarily in a synthesis of theory, it also seeks to directly address a practical problem through a practice-led approach. As such, the close study of a set of software-based artwork case studies from the Tate collection (the cultural organisation partner in this AHRC Collaborative Doctoral Partnership) form the core evidence base on which the research draws. The conservation of software-based art is a relatively new activity for museums, and has so far only received limited attention in research and published literature. This project represents the first major study of documentation within this emerging area of practice and may have applications in the wider field of software preservation, particularly for other kinds of software-based cultural work such as video games.

In **Chapter 1** I introduce the research topic and provide a rationale and methodology for its undertaking. As this project is interdisciplinary and uses terminology from several domains which may not be familiar to all readers, I first introduce and

disambiguate some key terminology to arrive at working definitions. Through a review of the state of the art in the field of software-based art conservation, I develop a rationale and scope for this research. I then describe the methodological approach this research has taken and introduce the six artworks which are discussed throughout this thesis as case studies and are a major source of evidence for the conclusions drawn.

In **Chapter 2** I explore what software is and how it is used as a medium, with the aim of identifying the challenges it presents as the object of conservation. I start by identifying some of the key technical characteristics of software and introduce a model for understanding the processes which occur within the realisation of a software-based artwork. This model posits that, while software might be seen as consisting of digital objects, the human experience of software can only be understood as a performance, during which these objects interact with a technical environment. I then explore the place of software in the history of art, identifying diversity in its usage and arguing that only some of these use types constitute what we consider software-based art. Building on the preceding sections, I conclude the chapter by identifying the medium-specific conservation considerations presented by software.

In **Chapter 3** I explore the nature of the document as a theoretical construct and a crucial part of conservation practice, with the aim of assessing the suitability of existing approaches to dealing with the medium-specific conservation considerations identified in Chapter 2. I begin by considering the development of documentation theory and discussing the potentially expansive notion of the document. I isolate some of the key principles in understanding the document in relation to the subject it documents, and the particular significance of documentation as something informational and representational. This is followed by an in-depth examination of the kinds of documentation found in conservation practice and a reflection on how they might need to be reconsidered in light of the characteristics of software-based art identified in Chapter 2. Three core documentation challenges emerge from this analysis, which are focused on in turn in the following three chapters.

Software is structurally complex and closely linked to the technical environment in which it is executed, and understanding and documenting these structures is crucial to the preservation of software-based artworks. In **Chapter 4** I consider how this information can be effectively derived and represented. I begin by framing software analysis and documentation in relation to elements of the conservation workflow and related concepts from software engineering. Building from a critique of the dominant

approach of source code analysis, I consider other complementary reverse engineering and software analysis techniques—particularly those which address software binaries and processes—in terms of their potential use in generating knowledge to aid understanding of the software performance. In the last part of the chapter I consider how these structures might be formally represented, particularly with information systems in mind. Comparing a number of existing metadata models from related domains, I find them unsuitable for this purpose and develop a conceptual model (expressed as an OWL ontology) for guiding the creation of human and machine-readable structured representations.

Changes to some of the components of a software-based artwork are expected to occur in their long-term preservation. In **Chapter 5** I consider how documentation might be used to ensure that the significant characteristics that constitute the core identity of a work are captured and appropriately managed through time as it is realised in different contexts. Dominant theoretical frameworks in digital preservation and art conservation, including the notion of significant properties, are examined and considered in terms of their practical applications. I introduce the idea of significant knowledge as an alternative view on this problem, and develop a set of knowledge categories for the software-based art domain. Finding there to be a need for a better defined approach to capturing identity at the level of the software performance, I introduce concepts from requirements engineering as a means of formalising the constraints on what a software-based artwork should do and how it should do it.

Software-based artworks are the result of processes largely unfamiliar to collecting institutions and are likely to continue to evolve through time while within their care. In **Chapter 6** I consider how the evolution of the artwork through time might be recorded by conservators. I introduce two contrasting approaches through which to conceptualise change, and theory from the study of software evolution which aids in understanding why software-based artworks experience change to varying degrees. I examine the nature of processes of creation and ongoing change in the life of a software-based artwork, at the software level, and how these processes might be understood and captured. I then consider how software might change in practice at two levels: the micro-level processes which are traceable through changes at the level of code and environment; and the macro-level decisions regarding the description of a particular transformation and the versioning of the software and artwork. Finally, I consider how we might describe the complex life histories of these works in narrative forms which consider the software-based artwork as something situated within a

broader socio-technical context.

Finally, in **Chapter 7** I conclude with an overview of the research contributions generated in the preceding chapters and a consideration of the potential limitations of the practical outcomes. I then reflect on some of the overarching themes identified within the thesis and present a set of recommendations for future research in the field.

1.2. Key Knowledge Contributions

This research has been undertaken in response to a need to develop solutions to challenges in an emergent and thus poorly defined problem-space. As such, it has not sought to respond to a single specific research question, but rather undertake work to better define this problem-space and construct pragmatic solutions to gaps identified using existing knowledge where possible. As a result, a set of interconnected but standalone research contributions have been generated, each responding to a specific knowledge gap. These are spread throughout the five core chapters of the thesis. In this section I present a concise overview of these contributions so that they can be located and consulted independently of the high-level narrative.

A group of these contributions were formulated specifically to improve the delimitation of the problem-space and can be found in Chapters 2 and 3:

- A **lexicon of clearly defined terminology for describing the medium-specific characteristics of software-based art**, each term being implicated in the challenges faced by conservators (p. 63)

- A thorough **examination of the potential scope of the 'document' concept within a practice software-based art conservation** (p. 98)

A second group of novel outcomes resulted from research in Chapters 4, 5 and 6, which focused on particular issues in software-based artwork documentation identified in prior chapters:

- An **extension of existing conservation approaches to software analysis to incorporate additional methods from reverse engineering**, particularly those which can be applied in the absence of source code (p. 116)

- A **conceptual model for the description of software structures and versions**, with potential applications in the extension of collection-related

information systems and metadata (p. 131 and p. 204)

- An **approach to the formalisation of the identity of software-based artworks** which emphasises categories of knowledge and documentation in place of sets of defined properties (p. 155)

- A **theoretical synthesis of conservation documentation and requirements engineering documentation**, with particular relevance to the documentation of works in which functionality is the primary purpose of software (p. 161)

- A **study of the phenomenon of software evolution in the lives of software-based artworks** and its implications for their effective documentation (p. 190)

- A **proposal for a practical approach to artwork biography for software-based art** and reflections on its connection with a practice of technical art history (p. 211)

While of value as independent research contributions, these outcomes form an interconnected framework which I propose might serve to support a more holistic practice of generating conservation documentation for software-based artworks, as it emerges.

1.3. Terminology

This research is by its nature cross-disciplinary, operating at an intersection between art conservation, digital preservation, computer science, information science and media theory. Readers of this thesis may therefore be from any of a number of different domains and as such familiar with only a portion of the technical language used, or only with particular uses of a term. The majority of specialist terminology is defined as it is introduced within the text, but some particularly fundamental definitions—and the ambiguities surrounding their use—are discussed in this section for the sake of clarity.

1.3.1. Conservation and Preservation

The terms *conservation* and *preservation* both occur regularly alongside each other (sometimes being used interchangeably) in the literature around the care of museum collections, particularly collections of artworks. We can find discussions regarding the

meaning of this terminology as far back as 1985, when Pamela W. Darling highlighted problems with the conflation of the two terms in the American Institute for Conservation of Historic and Artistic Works' (AIC) Abbey Newsletter (Darling, 1985). Darling also acknowledges the two words' respective roots in libraries and archives (for preservation) and museums (for conservation).

A full disambiguation of these terms is beyond the scope of this thesis, but a number of general distinctions are made. This first is the use of conservation to refer to the profession of conservation and its activities, as defined by the AIC:

> "The profession devoted to the preservation of cultural property for the future. Conservation activities include examination, documentation, treatment, and preventive care, supported by research and education." (American Institute for Conservation of Historic and Artistic Works, 2016)

Preservation is itself embedded within this definition, and as such I use the term to describe the *goal* of conservation. In addition, I use *digital preservation* to refer to a separate field, which the Library of Congress defines as encompassing "the active management of digital content over time to ensure ongoing access" (Library of Congress, 2012). Much as preservation has its origins in and has its origins in records management (Day, 2000). While it has distinct origins in traditional art conservation, the conservation of art with a digital component has become increasingly closely connected with the field of digital preservation.

1.3.2. Representation

The term *representation* is used in several slightly different senses in this thesis, all of which ultimately relate to either one or both of two primary meanings of the word[1]:

1. The potential for something to act on behalf of or in place of something else.

2. The depiction or portrayal of something in a particular way.

Within the text these uses are distinguished by context, so below I provide some examples of their usage to aid comprehension.

The first significant use of the term in this thesis relates to the technical characteristics

[1] These are derived from the subdivision of definitions presented in the Oxford English Dictionary definition of representation (anon. representation, n.1, 2018).

of software and digital data, all of which fall under definition 1. Source code and executable binaries are both particular representations of the same software program, with distinct uses: source code is human readable and writable; binary code is machine executable. This usage is discussed further in Chapter 2. Representation in this sense may also be used to describe some of the products of software and preservation processes, such as disk images. A disk image is a digital representation of what would traditionally have been the contents of a physical disk drive. The actual data content of a disk image is identical whether it is stored as a raw disk image file or on a physical drive. Similarly, a document (such as this thesis) might have multiple possible representations in different file formats. The use of representation in this context points to the need for *representation information* from which to correctly interpret these file formats, a component explicitly modelled within the dominant model of archival systems, the Open Archival Information System and discussed further in Chapter 3 (CCSDS, 2012).

The second significant use of the term representation is in relation to documentation, where the type 1 and type 2 definitions become intertwined. This is because all documents to some extent act on behalf of the thing they document and depict or portray that thing to some degree. A narrative description of an exhibition for example, is a depiction of that exhibition from a particular viewpoint. However, in portraying qualities of the installation, the reader of the narrative may form an impression of the work which stands in for the physical experience, despite not necessarily having seen the installation. In the case of a representation of a thing as machine-readable data, the documentation may act as a surrogate or stand-in for the artwork through, for example, structuring that permits actioning of preservation policies. This kind of representation, such as a metadata record, I refer to as *structured representation.* Representation in the context of documentation is discussed further in Chapter 3.

1.3.3. Software-based Art and Genre Terms

The term *software-based art* is one which has become increasingly widely used the art conservation field, while resisting formal definition. While a detailed examination of the meaning of the term and its relationship with overlapping terminology can be found in Chapter 2, a working definition is required for its usage prior to this discussion. The definition of software-based art used in this thesis is: *art for which software is the primary artistic medium*. Taking a constitutive meaning of artistic medium (e.g. a sculpture in the medium of bronze or a drawing in the medium of pencil), this definition would pertain to artworks where software is the primary

mechanism in the realisation of the work and the primary material which the artist has chosen as a means of expression. This usage has its origins in discussions at Tate[2] in 2010 around the refinement of language to describe such works in its collection.

It is important to note that software-based artworks may incorporate other artistic media, limited not just to computers and other electronic equipment, but perhaps including sculptural elements or precisely defined installation environments. While I discuss such physical considerations where necessary in relation to a particular artwork case study (particularly as part of the conceptual whole of the artwork) the focus of this thesis is primarily on the software (and to a lesser extent, hardware) components, as these are what makes software-based art unique and demanding of particular conservation consideration. A distinction is consistently made between the software and the artwork in the text.

I generally avoid the use of other genre terminology (such as new media art or software art) to refer to software-based art, unless making reference to specific historical movements or trends with which a particular artwork might be associated. The only other art genre terminology that will be used more frequently is *time-based media*, which Tate defines as "'works of art which depend on technology and have duration as a dimension." (anon. Conservation – time-based media, n.d.). This is useful as a higher-level grouping of software-based art with other types of art with similar time-based characteristics, which together typically fall within the care remit of the same conservation team within a museum. Software-based art should not be confused with the distinct *software art*. The latter, as Christiane Paul clarifies in *Digital Art*, is closely linked to the tradition of software artists engaging directly with coding and the formal languages of computation (Paul, 2015). While all software art would fall within the classification of software-based art, the inverse is not true.

1.4. Literature Review

This research operates at the intersection of two disciplines—art conservation and digital preservation. Despite distinct origins (see Section 1.2.1), the two have become increasingly enmeshed as conservators of time-based media artworks have sought

[2] While there is no documentation of these discussions, they are evidenced by the use of the term in a number of Tate linked research outputs from between 2010 and 2014 (Laurenson, 2010, Falcão, 2010, Falcão, et al., 2014) and its adoption as a term in collections management systems.

to deal with the challenges posed by digital materials entering museum collections. The conservation of software-based art has emerged at this nexus. In this section I introduce key literature from art conservation (primarily in relation to time-based media) and digital preservation, and review the current state of theory and practice surrounding the conservation of software-based art. In addition to identifying gaps in existing literature, this initial review also serves to position the approach taken in this research in relation to existing perspectives on conservation. The need for the synthesis of new knowledge from other disciplines necessitates the introduction of material from other bodies of literature at later stages of research. These are introduced and discussed within specific chapters, most significantly: media theory in Chapter 2, documentation theory in Chapter 3 and various aspects of software engineering in Chapters 4, 5 and 6. The search strategies adopted in identifying key literature are detailed in Appendix IV.

This research focuses on the practice of conservation that has emerged around museums that care for collections of art, where it developed in response to traditional modes of practice such as painting and sculpture. Salvador Muñoz-Viñas suggests that the approach of conservators during the early, “classical” era of conservation theory might be best understood in relation to the principles of scientific conservation (Muñoz-Viñas, 2004). This, he suggests, is an approach to conservation driven by “strong, implicit principles”, which centre on the notion of an artworks “true nature”, understood as residing in its constituent materials, and best maintained through objective modes of scientific enquiry and treatment (Muñoz-Viñas, 2004, p.90). Problematising these principles and proposing an alternative, pragmatic and socially-situated perspective on conservation theory, Muñoz-Viñas’ critique is emblematic of a fundamental shift in thinking which has occurred over the past few decades and is evidenced in much of the theory which has emerged from scholars of conservation over the past few decades. This includes the young sub-discipline of *time-based media conservation,* which concerns the care of artworks with a technological component which unfold over time (Tate, 2017)—and encompasses software-based art.

The variable, ephemeral and changeable nature of such artworks has brought to the fore an array of philosophical, ethical and practical considerations in sustaining such artworks through time—and in response, a growing body of research seeking to address them. Foundational knowledge was cultured in early symposia. In 1997 participants in *Modern Art: Who Cares?* in Amsterdam grappled with the challenges

of modern materials, including electronic media (INCCA, n.d.), while in 2000, *TechArchaeology: A Symposium on Installation Art Preservation* was held at San Francisco Museum of Modern Art and explored the preservation of technology-based installation artworks (Real, 2001). In the 2000s we see a host of institutionally-led research projects exploring aspects of time-based media conservation, including *The Variable Media Initiative* (Depocas, et al., 2003), *Capturing Unstable Media* (Fauconnier, & Frommé, 2003), *Inside Installations* [2004-2007] (Scholte, & Wharton, 2011), and *Documentation and Conservation of Media Arts Heritage* (or DOCAM) [2005-2010] (DOCAM, n.d.). In common to these projects was an acknowledgment that instead of being fixed and centred on specific material artefacts, time-based media artworks have the potential to vary in their constituents between realisations and may possess medium-independent characteristics. Through focused research on case study artworks, these projects explored the ways in which change might be negotiated in the care of time-based media artworks, including, in some cases, software-based artwork case studies.

In parallel, conservators embedded in museums with collections of time-based media art were beginning to formalise some of these ideas. Pip Laurenson proposed a theoretical model for approaching the conservation of time-based media based on experiences at Tate, which formalised the distinction between an artwork and its ongoing realisation through time as variable “installed events” (Laurenson, 2006). Elements of this theory were operationalised by Joanna Phillips, in a documentation model used at Solomon R. Guggenheim Museum, which distinguished between a documenting a work’s identity and its iterative staging through time as display equipment changed (Phillips, 2007), while the Matters in Media Art consortium later developed a set of guidelines and templates with similar ambitions (Matters in Media Art, 2015). Underlying theory has remained under question however, with growing bodies of research engaging with ideas of intentionality and authenticity in relation to the conservation of modern and contemporary art (van de Vall, 2015, Wharton, 2016) and the changing role of the museum in relation to restaging performance and installation artworks (Wharton, & Molotch, 2009, van Saaze, 2013, Laurenson, & van Saaze, 2014). These might be considered as emblematic of what Hanna Hölling identifies as a new *relativistic* approach to conservation; a shift from a practice focused simply on prolonging artworks material forms, to conservation as a “complex techno-cultural practice with a strong, retroactive impact on its objects and subjects” (Hölling, 2017, p.89). While this new approach offers fresh perspectives on the potential role of the conservator, it remains somewhat detached from the pragmatic

concerns of the time-based conservator attempting to care for a growing collection of software-based artworks, for which even the “techno” component of this practice remains poorly understood.

Before discussing the state of the conservation of software-based artwork in more detail, I will introduce the second discipline on which it depends: digital preservation. This is a relatively young, techno-centric discipline which has emerged in parallel to time-based media conservation. While early examples of literature pertaining to the preservation of electronic records can be identified as far back as the 1970s (Day, 2000), it is around the turn of the 21st century that we see digital preservation at a point of coalescence. Here we find the literature setting out the issues that would occupy the field for the coming years: the loss or failure of the media on which it is stored and the process of technological obsolescence which renders it inaccessible or unreadable (Rothenberg, 1995, Waters and Garrett, 1996, Chen, 2001). In the years following we see a response which, rather unlike contemporary theories of art conservation, is more focused on the development of standards and tools that could guide institutions seeking to establish systems and policy for digital preservation. These have nonetheless gradually worked their way into art conservation practice over the past decade—as reflected in a growing body of practical guidelines (Matters in Media Art, 2015, Digital Preservation Coalition, 2015, Fino-Radin, 2018). As a result, many of the fundamentals of the long-term preservation of digital materials in art collections, such as methods for establishing secure archival storage, are now relatively well understood and surmountable providing appropriate technological and organisational frameworks are implemented.

While such developments have benefited the preservation of various forms of digital media—by maintaining the integrity of the ones and zeros of digital information—the problem of ensuring long-term access to the content that the bits represent is a much harder problem to solve. In 2001, Howard Besser’s examination of “electronic art” preservation identified a number of challenges presented by digital media artworks, including difficulties in identifying their boundaries where they extend into the surrounding technical environment (“the inter-relational problem”) and their complex relationship with the technologies used in the playback of their stored form (“the translation problem”) (Besser, 2001). Both implicate the obsolescence-induced precarity of the complex systems constituting and surrounding such artworks as a significant risk to continued access. For media types which are relatively clearly bounded, such as digital video, considerable progress has been made towards

understanding these issues. Detailed technical guidelines for preservation are in the process of being established (IASA Technical Committee: Standards, Recommended Practices, and Strategies, 2018), while recent research has begun to isolate granular issues such as achieving consistent playback (Rice, 2015). For media with less clear boundaries, such as software, our understanding remains in a rather less developed stage.

Nonetheless, addressing the conservation of software-based art has become a practical need for museums and other collecting institutions over the past decade, and the foundations for a specialised area of practice have begun to emerge. With important initial discussions occurring in events organised by universities and libraries (Konstantelos et al., 2012, National Digital Information Infrastructure and Preservation Program, 2013), the first museum-led events dedicated to the issue soon followed: *Technology Experiments in Art: Conserving Software-Based Artworks* in Washington in 2014 (Time-Based Media and Digital Art Working Group, 2014), followed by *TechFocus III: Caring for Software-based Art* in New York in 2015 (Electronic Media Group, 2015). While tenets of digital preservation such as fixity and redundant storage remain relevant to software-based art, a number of points of divergence from file-centric approaches have emerged. Studies exploring the medium-specific qualities of software have found that they tend to exacerbate such risks in the face of certain technological change, resulting in both a faster onset of obsolescence and a complicating of the identification and effective treatment of risk factors (Falcão, 2010, Fino-Radin, 2011, Laurenson, 2013). This body of research particularly emphasises the significance of the connection between software and its technical environment both locally (on operating systems and supporting software) and as it extends into external services and data accessed through the internet—although stops short of offering solutions.

In another point of divergence from the preservation of media such as digital video—which has been largely focused on file format migration—research on preservation strategies for software-based art has favoured emulation. Emulation was originally proposed by Jeff Rothenberg as a means of bypassing the continual “heroic effort” demanded by migration (Rothenberg, 1995, Rothenberg, 2002), and proposes that access to digital materials be maintained through the use of a layer of software which translates instructions designed for one (obsolete) system into those understood by another (contemporaneous) system. While Rothenberg’s proposal was criticised at the time on the grounds of its focus on preserving functionality over the content

represented by digital information (Bearman, 1999), when applied to software, these criticisms hold less weight. Functionality, or the “ability of software to ‘do’ something”, is, as Laurenson points out, one of the defining characteristics of software-based art (Laurenson, 2013). We have thus seen a renewed interest in emulation over the past decade, alongside the maturation of the tools required to implement it, and a number of compelling demonstrations of its viability as a tool in the conservation of software-based artworks (Lurk, 2008, Lurk, et al., 2012, Rechert, et al., 2013, Falcão, et al., 2014). It is important to note that this ongoing engagement with emulation does not preclude the value of other approaches. Although examples of published work on the migration of software-based artworks are few, it has recently been demonstrated to be an effective strategy for conserving internet artworks (Phillips, et al., 2017). In a more radical departure from established approaches, it has been suggested that in some cases accepting a degree of loss in the process of change might be necessary, and that this might even serve to highlight the historical significance of technological change (Guez, et al. 2017).

While the research highlighted above has undoubtedly pushed forward our understanding of the conservation of software-based artworks, there is a noticeable gap in the literature in relation to documentation. While several documentation-centric projects pertaining to time-based media conservation have produced templates for describing particular realisations of artworks (Phillips, 2007, V2_Institute for the Unstable Media, 2003, DOCAM, n.d., Matters in Media Art, 2015), their suitability for a medium which was relatively poorly understood at the time of their formulation makes their value difficult to assess. Documentation of artwork identity lacks even generic templates such as these. Despite an interest in certain frameworks such as significant properties as a means of capturing such information, both within the conservation of software-based art (Laurenson, 2013) and software preservation (Matthews, et al., 2008), there is little evidence of their use in practice. Recent commentary has suggested that this relates to ambiguity in the definition and application of the significant properties concept (Dappert, & Farquhar, 2009, Yeo, 2010), and further work is required to understand whether the concept might be operationalised in the preservation of software-based art, particularly in relation to a relativistic perspective on conservation activities. Capturing documentation on the nature of the changes (both material and conceptual) that occur during an artwork’s life are even less well understood—likely due to the nascent status of the previously highlighted forms of documentation, from which it logically follows.

While the literature on the documentation of software-based art remains small, there are two notable strands of recent research. The first draws on the established field of software engineering, seeking to reframe its principles within art conservation as a tool for analysis and documentation (Marchese, 2011, Marchese, 2013, Engel, & Hellar, 2014, Engel, & Wharton, 2014). Engel and Wharton's research on source code documentation shows particular promise in its application to real collections. The authors worked with a group of students at New York University to carry out the analysis of source code for a number of software-based artworks from the collection of the Museum of Modern Art (Engel, & Wharton, 2014). While the paper clearly demonstrates the power of this approach, questions remain as to how practical this kind of time-intense, specialised work is in relation to the limited resources of many institutions, and what other methods of analysis might be utilised for works for which source code is not available. Further work by Engel and Wharton suggests an emerging practice of technical art history for software-based artworks may also build on source code analysis (Engel, & Wharton, 2015), while evidence from other authors indicates that the close analysis of compiled software may also offer insights (Adang, 2013). The conservator has traditionally had an important role in investigating the material and process histories of artworks within the history of scientific conservation (Hermens, et al., 2012), yet how conservators might engage with or produce technical art history for software-based artworks in practice remains relatively unexplored.

A second strand of research can be identified in work that has sought to extend the documentation of software beyond immediate technical concerns, and instead consider the capture of contextual information. This has been explored particularly in the preservation of video games (McDonough, et al., 2010, Lowood, 2013, Kaltman, et al., 2014), where interactivity is key, and where the inherent ephemerality of experiences (such as networked virtual worlds) prevents their stabilisation in any material form. Related issues have been explored for internet art by Annet Dekker, who develops a processual model for understanding the conservation of internet art and which touches on many issues relating to documentation (Dekker, 2014). This work has implications for understanding the ontology of an important category of software-based artworks for which the identification of boundaries is challenging. Even more significantly though, it points to the importance of documentation of context and process as a means of establishing meaning for a medium likely to experience change during the life of a work. Problems remain in how such research might be operationalised by conservators, however, given the lack of formalised models, and further work is required if we are to understand what kind of

documentation materials might support this.

In summary, pragmatic strategies for the documentation of software-based artworks remain poorly defined, as evidenced by a small body of literature devoted to its challenges. Despite an increasingly sophisticated theory of conservation, alongside a powerful set of technical tools, a practice of software-based art documentation has been slow to emerge. A number of factors seem to have contributed to this. At a fundamental level, there appears to be a lack of agreed upon terminological frameworks for describing software-based artworks and their technical constituents and the way in which they might be addressed and analysed. Furthermore, identifying the significance of these constituents in relation to the artwork's identity is fraught with challenges regarding the ontology of the work, and would benefit from further exploration in reference to real case studies. Finally, as strategies for preserving software-based artworks have remained emergent, defining a practice of documentation to support them has been difficult. With methods recently becoming more established however, it seems like an appropriate time to revisit documentation theory and reconsider what documentation might mean when supporting the conservation of software-based artworks.

1.5. Rationale and Scope

In the previous section I identified a lack of practical and theoretical frameworks to guide the creation of conservation documentation that might effectively support the long-term care of software-based artworks. This research is concerned with addressing this gap, and the first consideration is doing so is the identification of an appropriate approach. Ultimately this research topic is inextricably linked to an area of professional practice—art conservation—and so it seems immediately obvious that this research should seek to contribute to this through the new knowledge generated if possible. This points to the significance of a practice-led approach, which—as opposed to a practice-*based* approach which would seek to carry out practice *as* research and present the outcomes as original contributions—engages with practice closely but focuses on creating original contributions through empirical research. It is important to acknowledge how this impacts the dimensions of conservation theory engaged with in this research. Most significantly, this research is limited through a reliance on *existing* modes of practice—primarily occurring within museums—and neither proposes nor extensively engages with approaches which are radically divergent from this established perspective. Therefore, some elements of the research will be normatively framed, and grounded in the assumption that the

principles of a Brandian[3] approach to conservation require modification rather than reinvention. This is justifiable at this early juncture in the development of a practice of software-based art conservation as we cannot, after all, hope to reinvent a field until it is understood in relation to the suitability of existing modes of practice.

This research will therefore aim to develop research contributions with practical implications for conservators of software-based art through an interrogation of the characteristics of the medium and aspects of professional conservation practice. Two fundamental theoretical gaps must be addressed initial, both of which serve to better define the problem-space that is being addressed. The first, which is addressed in Chapter 2, is an incomplete understanding of software as a medium—an ontologically sound and scientifically grounded understanding of which is essential to its conservation. The “significant difference” of software-based art, as Laurenson puts it (Laurenson, 2013), is clear, but there remain questions over the nature of this difference and how the technical characteristics of software might impact the way in which we approach their conservation. Furthermore, there is need for further investigation into the ways in which software can be used by artists (in terms of their intent regarding the work) and how these might affect a potential treatment of the media.

The second theoretical gap, which is addressed in Chapter 3, is a limited understanding of the body of documentation which might support a software-based artwork’s long-term preservation. Materials termed documentation might include a multitude of descriptive and representational materials which are linked to a museum object, event or other recorded phenomenon; such flexibility is desirable in dealing with variability among artworks. However, while conservation documentation is composed of multifarious documents and is unlikely to conform to any one standard, well defined approaches may still provide a important baseline and means of achieving best practice. Therefore, it is important we understand the purpose of documentation in relation to the conservation activities that occur in the care of a software-based artwork. There are approaches from the software engineering and computer science domains which may be well suited to fill some of the gaps in this area, and an attempt to consolidate these with art conservation approaches (a focus

[3] Pertaining to the theories of conservation developed by Italian theorist Cesare Brandi, whose place within the history of the theory of contemporary art conservation is introduced by Hölling (2017).

of this thesis) begins with the specification of a conceptual framework for documentation theory and practice. Existing approaches also require consideration, as suitable frameworks may well already exist given the several decades of research and practice within the time-based media conservation field.

There are several areas which relate closely to the aims of this research, but for practical reasons (relating both to the expertise of the author and time constraints) must be considered out of scope. Software-based artworks may incorporate physical components and so bring with them concerns over their gradual degradation and eventual loss. They may also involve elements of performance and so require consideration of staging and scoring, or the need for installation and so require careful consideration of lighting and display equipment. While these might be important considerations when addressing the conservation of a software-based artwork holistically, they are considered largely out of scope of this thesis, in order to restrict focus to addressing software and its unique challenges. Exceptions are made where referencing these considerations is important to the overarching concept and artistic intent of a piece, and for computer hardware as a physical component, as it is inextricably linked to software.

Finally, it is important to note that this thesis is generally based on the premise that the software-based art conservator (and the reader of this text) has not received a higher education qualification in computer science, and as such terminology from this domain is clearly defined throughout. While training in these areas may become more commonplace among conservators in the future, such explanation and terminological synthesis is important during a time of transition within the field. Collaboration with computer scientists has been a recurring theme of recent research in the conservation of software-based art (e.g. Engel, & Wharton, 2014, Dover, 2016, Rechert, et al., 2016), particularly within museums, and is likely to remain an important and necessary activity. The focus of this thesis lies in identifying the elements of technical work which might form part of the conservator’s remit, while highlighting parts of the process which may demand connection with software specialists.

1.6. Methodological Approach

In this research I have applied a hybrid methodological approach which combines *constructive research* and *case study research*. A constructive research approach was chosen as it is particularly well suited to research which seeks to develop solutions to real-world problems and is designed to connect practical problems with

existing theory (Lehtiranta, et al., 2017). Gordana Crnkovic, an advocate of constructive research methods within software engineering, defines the approach as follows:

> “Constructive research method implies building of an artifact (practical, theoretical or both) that solves a domain specific problem in order to create knowledge about how the problem can be solved (or understood, explained or modeled) in principle. Constructive research gives results which can have both practical and theoretical relevance.” (Crnkovic, 2010, p.4)

The construction of an “artifact” is also the goal of this research—in this case a pragmatic framework, grounded in relevant theory, to guide the application of appropriate analysis and documentation methodologies to the conservation of software-based art. Constructive research methodology is closely related to design science research methodologies (Dresch, et al., 2015) which similarly seek to explore how research may contribute pragmatic solutions rather than focus on explaining phenomena.

Other related methodologies suitable for a practice-led research approach were considered as an alternative to constructive research. Grounded theory, while similarly fostering the iterative construction of theory alongside analysis (Bryant, & Charmaz, 2007), was rejected due to its focus on explanatory theory production over practical outcomes. Action research was also considered due to its applications in research that aims to solve real-world problems (Stringer, 2013). However, the methodologies focus on addressing the study of social groups and organisations makes it unsuitable for application to the problem identified early in this: a lack of knowledge about the technical characteristics of software as a component of software-based artworks, their significance and the methods that might be used to describe them. As discussed in Section 1.5, not taking such an approach excludes the dimensions opened by relativistic conversation theory (as discussed in Section 1.4), particularly in relation to networks of care which may surround complex artworks both inside and outside the institution. While this certainly excludes a potentially interesting avenue of research, this can be justified at this formative point in the development of a practice of software-based art conservation due to the need for a well-defined technical basis on which to understand the medium in question.

Constructive research methodology has already found use in research from varied domains including digital preservation (McGovern, 2009) and computer science (Crnkovic, 2010) and as such suitable models for its use in this research already exist.

Building on a methodology developed by Kasanen, Lukka and Siitonen in a management research context (Kasanen, et al., 1993), Nancy McGovern applies a constructive research approach to a digital preservation scenario and the development a conceptual model (McGovern, 2009). I reuse the core of this constructive methodology here, which McGovern characterises as having the following stages:

1. “Find a relevant practical problem with research potential
2. Obtain a general and comprehensive understanding of the topic
3. Build an innovative solution (or construct)
4. Demonstrate that the solution works
5. Show the theoretical connections and research contributions of the solution
6. Examine the scope of applicability of the solution”

(from McGovern, 2009, p.64)

Stage 1 has been addressed in the 1.3. Rationale and Scope section in this chapter. Chapter 2 and 3 build a comprehensive understanding of the topic, so addressing Stage 2. The result of these two chapters will be referred to as a *conceptual framework*, which can be defined as “the system of concepts, assumptions, expectations, beliefs, and theories that supports and inform your research” (Maxwell, 2005, p.39). This conceptual framework incorporates a study of the use and potential significance of software as a medium, and the implications of this material choice for conservation (in Chapter 2). The other part of the framework, incorporating knowledge generated from the first, is an examination of documentation theory and its connections with documentation practice in art conservation (in Chapter 3).

Due to the close connection of this research to conservation practice, interviews with those engaged with professional activities or research projects related to the care of software-based artwork collections were undertaken, to help further refine the conceptual framework. The aim of these interviews was to develop a richer understanding of an area of study which, as practice-driven, is not always able to publish with the frequency of a traditional academic disciplines. The interviews were undertaken using a semi-structured approach and were designed to gather respondents’ perspectives and priorities relating to the documentation of software-based art. These interviews are not a core part of the research methodology

employed here, and could not be considered comparatively as part of a qualitative analysis. Rather, they act as an extension to the literature review underpinning the development of the conceptual framework in Chapters 2 and 3. The individuals interviewed were: Deena Engel, Ben Fino-Radin, Mark Hellar, Joanna Phillips, Klaus Rechert, Eric Kaltman, Jon Ippolito and Gaby Wijers. Ethical issues relating to interviews and informed consent are discussed in Section 1.6.1 below.

Stage 3 of McGovern's constructive research methodology is addressed in Chapters 4, 5 and 6. Here, specific problem areas as identified in the conceptual framework chapters, are addressed through the construction of appropriate solutions through the reframing and extension of existing theoretical frameworks from relevant domains of knowledge. These solutions are then tested for their compatibility with practice (fulfilling Stage 4) within the relevant section, using evidence from a set of case study artworks from the Tate collection. The data that provides this evidence is derived from the in-depth study of these case studies using a combination of direct technical analysis at the software and hardware level, examination of secondary materials (such as existing documentation and archival materials) and research into their production and material histories. These case studies are referred back to continually throughout the text, and to ensure a basic understanding of the artworks it may be useful for the reader to consult the summary descriptions and images in Appendix I.

This research was original formulated as a collaboration between King's College London and Tate, and was predicated on the opportunity to work directly with the latter's collection in addressing the research questions formulated. As such, this research is a direct response to challenges currently faced by conservators at Tate. There are currently ten software-based artworks in the Tate collection, from which a set of six were selected based on their diversity in technological platform, construction and behaviour (and so being meaningfully comparable and somewhat representative of the diversity of the medium). The other four artworks share characteristics and were determined to be unlikely to provide sufficient additional insight to warrant detailed in-depth study. The complete set of case selected are presented in Table 1 below.

Title	**Artist**	**Year created**	**Technical characteristics**	**Operating system**	**Core technologies**
Becoming	Michael	2003	Wall mounted monitor displaying	Windows XP	Shockwave Director and

	Craig-Martin		dynamic 2D assemblage		Lingo script
Brutalism: Stereo Reality Environment 3	Jose Carlos Martinat Mendoza	2007	Sculpture with mounted printers and web search software	Linux (Ubuntu)	Java and MySQL
Colors	Cory Arcangel	2005	Video processing software	Mac OS X	Objective C / C++
LiMac Museum Shop	Sandra Gamarra	2005	Actively maintained website with online shop	Linux (CentOS)	MySQL, PHP, HTML and CSS
Sow Farm (near Libbey, Oklahoma) 2009	John Gerrard	2009	Real-time 3D simulation	Windows 7	Quest3D and HLSL
Subtitled Public	Rafael Lozano-Hemmer	2005	Interactive installation	Windows XP (via Mac OS X Bootcamp)	Borland Delphi

Table 1. List of the software-based artwork case studies examined in this thesis, with basic descriptive information for each. See Appendix I for further descriptive details.

The case studies are integrated with the constructive research methodology, primarily through their use in demonstrating the viability of proposed solutions at Stage 4 in the methodology. Solutions and strategies developed during Stage 3 are tested against case studies in each case and are presented as supporting evidence through detailed account embedded within the relevant section of each chapter. The majority of these artworks were already well studied prior to this research—research by Pip Laurenson, Patricia Falcão and others at Tate precedes mine, and generated a considerable amount of documentation and insight. Both their documentation and their first-hand accounts of experiences with the works has considerably informed my examination. In some cases, and where a gap was identified in existing documentation, artists (and sometimes their collaborators) were consulted or interviewed regarding specific issues and questions.

In the final chapter of the thesis (Chapter 7), I discuss the overall research

contributions to theory and practice (Stage 5) and reflect on the wider applicability of the framework developed (Stage 6).

1.6.1 Legal and Ethical Considerations

For artwork related materials (understood here in relation to sets of physical and digital components), access was granted in accordance with institutional policies on research using collection materials, including my own abidance by Tate's Code of Good Practice in Research. In cases where the physical components associated with artworks were accessed, this was always carried out in collaboration with time-based media conservation staff at Tate and steps taken to ensure that such interactions would minimise impact on the objects. For all digital materials access was granted only on secure workstations within Tate property using temporary research copies. No interventions or treatments were carried out on any artworks or associated materials during this research.

In examining artwork and documentation materials typically closed to general audiences, ethical considerations were raised by this research in relation to the intellectual rights of the artists whose works were examined. Such materials were not shared with others during the research and, as stated above, security was ensured by accessing materials only on Tate property using secure workstations. Some information or data derived from the analysis of artwork materials is incorporated into this thesis as evidence, as are several source code fragments. Where these uses have occurred, they will be approved with the artist (or their representative) prior to general access being granted through thesis deposit/publication or otherwise redacted. Additional considerations were raised through engagement with techniques for software reverse engineering, which could reveal information that artists had not intended to share. Where reverse engineering tools were employed during this research, they were used only in cases where materials equivalent to those being reverse engineered were already accessible as part of the artwork's documentation, or where they did not compromise intellectual property.

For the interview series, ethical approval was gained from King's College London's Ethics Review. In each case, consent was granted by all participants that their responses could be used in the context of this research and a signed consent form stored. The option of requesting that data not be used beyond this project was also offered and will be respected for those individuals.

CHAPTER 2

SOFTWARE AS MEDIUM AND MATERIAL

2.1. Chapter Outline

In Chapter 1 I identified the need for a technically informed understanding of the use of software as a medium and material of conservation concern, as the first part of a conceptual framework for further refining the problem space this thesis seeks to address. I also presented a working definition of software-based art as 'art for which software is the primary artistic medium'. In this chapter I explore the two key concepts in this definition—the characteristics of software as a material (defined here simply as the substance of software) and its significance as an artistic medium (defined here as something which is used as a means of artistic expression)—and consider how they might together impact conservation. I clarify both definitions further within the chapter.

Despite its limited treatment within art conservation, the study of software has considerable precedent from across a number of disciplines—therefore this chapter takes an approach of robust review and synthesis of existing literature. It also incorporates information gathered during the examination and analysis of a number of the case study artworks. In the first portion of the chapter, I examine the technical characteristics of software, taking a bottom-up approach isolated from concerns related to artistic use. Based on these findings, I develop a model of software

performance which generically describes the process that occurs behind the experiential qualities of software. In the next section I explore the ways in which software has been used by artists, how they relate to specific genre terminology and our conception of software-based art as a category of artistic works. Building on the knowledge developed in the previous sections, I then consider how the unique qualities of software as a medium and material choice may impact our attempts to conserve it.

2.2. What is Software?

Definitions of software found in art conservation and software preservation literature are various and at times confused. Software might be talked about as a means of rendering other digital objects (for example, video player software to play back a digital video file), but in other cases software is itself the digital object of concern. Code is also a frequently referenced concept, yet this term has multiple related meanings within computer science. In this section I will explore the meanings of these terms and connect software as an observable phenomenon with its underlying technical foundations—a process which, I propose, will elucidate important characteristics of software as a material of conservation concern.

While this section deals with well-understood concepts within the computer science domain and attempts to generalise them, it also takes a perspective on software which is coloured by the cultural heritage context of this research. There are a multitude of other perspectives on software. Software engineers for example, may consider software as a product to be designed, developed and packaged, in order to solve a problem. Mathematicians on the other hand, might approach software as a logical construct, understood within computational theory. The particular viewpoint taken here is that of a conservator engaged with the care of a cultural heritage collection. This perspective is ultimately experience-centred—software is considered a phenomenon which has been experienced and potentially could be experienced in the future.

2.2.1. Defining Software

The definition of software in Butterfield and Nogondi's *Dictionary of Computer Science* presents a pragmatic starting point for this discussion and its length permits the clarification of a number of important concepts. I consider this definition in three parts. The first of these parts defines software as:

> "A generic term for those components of a computer system that are intangible rather than physical." (Butterfield, & Ngondi, 2016)

This definition highlights the non-physicality of software: it is not a phenomenon that we are able to touch directly. This points to the significance of interface and the layers of abstraction through which humans interact with the physical layer of computer systems. This idea is important in understanding the relationship between software as experience and software as process—something I return to in Section 2.2.3.

The use of the word *intangible* to characterise software is also problematic, in that while it is a necessary condition, it does not offer sufficient contrast with other kinds of digital object that we might not consider software. After all, any digital object—be it a plain text file containing ASCII values or a JPEG raster image—might be considered just as intangible. This observation hints at an underlying ambiguity in the relationship between software and data, which can make drawing a clear distinction challenging (Suber, 1988, Oberle, et al., 2009). Software might require data sources in its operation and in some cases might be seen as part of the stuff of software—for example, the database underlying a collections management system or the graphics assets which make up a game environment. At the same time, software might be viewed as data—the code that makes up software is stored as discrete binary values in much the same way as any other digital object. Both are valid viewpoints—therefore, coming to a workable distinction between software and data comes down to selecting an appropriate level of granularity at which to work.

The second part of the definition helps with this selection by introducing a slightly more specific definition of software:

> "It is most commonly used to refer to the programs executed by a computer system as distinct from the physical hardware of that computer system, and to encompass both symbolic and executable forms for such programs." (Butterfield, & Ngondi, 2016)

This statement provides a basis for software as a countable digital thing by introducing the idea of software *programs*, which Butterfield and Ngondi define as a "set of statements that [...] can be executed by a computer in order to produce a desired behaviour from the computer" (Butterfield, & Ngondi, 2016). The software program is the level at which I will primarily address software within this thesis. In accordance with the ontological model of software proposed by Oberle, Grimm and Staab (Oberle, et al., 2009), software will be considered itself a subtype of data,

distinguishable through its potential to manifest as a sequence of computational activities and itself manipulate data. As with Pressman and Maxim's definition of software (Pressman, & Maxim, 2014), it may also incorporate non-program data where it forms part of the operation of the software.

Several other important concepts are introduced in this part of the definition: software programs are *executed* by a computer and they have multiple "forms" or *representations*. If programs are executed—that is, read and acted upon by a computer system—they must therefore be in an otherwise latent form, until they are called into action by whatever agent is able to trigger them. This latent form consists of encoded instructions, which the host computer system is able to interpret and act upon in some way. The symbolic form referenced in the definition is a representation of the software which the host computer system is not able to interpret or act upon, such as source code. The importance of this distinction and the transformation between representations is discussed further in Section 2.2.2.

This part of the definition also makes clear a further distinction between software and *hardware:* the physical components that make up a computer system. This distinction poses its own ontological challenges. Hardware components often contain deeply embedded software, known as *firmware*, without which they would be rendered non-functional. This kind of software is hard to separate from its specific physical carrier. Furthermore, hardware can be replaced with software through processes such as emulation. There has been historical debate over the validity of the distinction between software and hardware among philosophers of computing (Moor, 1978, Suber, 1988, Duncan, 2009). In this thesis, I adopt Duncan's position that the separation is valid when framing software programs as a unit of grouping for computational functions which are actualised by computing hardware (Duncan, 2009). This position accommodates firmware and allows for hardware to be non-physical, in cases where it is emulated.

The final part of the definition reveals a typological distinction within software:

> "A distinction can be drawn between systems software, which is an essential accompaniment to hardware in order to provide an effective overall computer system (and is therefore normally supplied by the manufacturer), and application software specific to the particular role performed by the computer within a given organization." (Butterfield, & Ngondi, 2016)

This highlights the separation of custom software designed to carry out a specific

purpose (application software) from that which forms the computational environment essential for supporting it (system software). The notion of environment is crucial to understanding the requirements for the long-term preservation of software, as their reconstruction provides the contingencies necessary to enable its successful execution. These ideas are developed within a conceptual model of software performance in Section 2.3.

2.2.2. Software Representations and Opacity

In the previous section, I introduced the idea that software can exist in multiple possible representations and introduced the fundamental distinction between source code (a symbolic representation) and executable representations. Source code typically refers to the human-authored expression of a software program, symbolically expressed using syntactically valid language but not directly executable by a computer processor. This source code can be transformed into something executable by a processor through the process of compilation, or through the action of an interpreter (an additional software component) which converts the source code into machine actionable instructions on-the-fly. In practice, source code is not the only component in the complex processes involved in the creation of software, which can involve the use of development environments, automation and reusable third-party components. Within this thesis, I will collectively refer to these as *source materials*.

Executable programs (sometimes called *binaries*) are the transformed, machine actionable products of source code (or source materials)—now represented in a form in which a computer processor can carry out operations based on the encoded instructions. Executable programs are also made up of code, but this representation takes the form of a lower level language designed for machine execution rather than human readability. This may be machine code (which is encoded in binary) at its lowest level, or in other cases an intermediate representation (such as bytecode) which requires interpretation by supporting software in order to be executed (e.g. Java or PHP) (see Figure 1 below). In practice, compiled software is not necessarily made up of just executable code—the code may also be accompanied by data, libraries and other components which are called upon as the program executes. For convenience, I will refer to this collection of digital components as a *software super-object*[4].

[4] From the Latin root of super, meaning *above* or *on top of*.

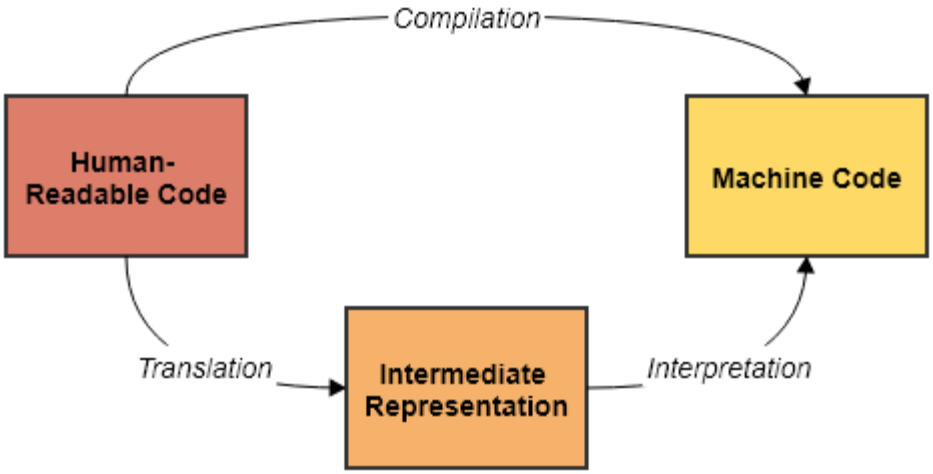


Figure 1. Diagram of transformations between software representations, indicating the potential for code to be compiled to machine code or an intermediate representation which must then be interpreted.

The structure of the software super-object which results from the process of compiling source materials varies considerably. In some cases all of the required functionality is packaged within a single executable file, which can then be run on the target platform with little additional configuration. In some cases, features and requisite data may be distributed among a number of files within an application directory. For more complex software, additional supporting software must be managed alongside the program and correctly configured within the compilation (or execution) environment for the software to function. Taking libraries (packages of resources another program can utilise in its execution) as an example: *static* libraries are accessed during the development of the software, and the necessary parts incorporated into the binaries when the software is compiled. *Runtime* libraries on the other hand, are accessed on-the-fly as the software is executed and must be included in the package of binaries.

Compilation is usually carried out with a particular platform (typically an operating system) in mind—therefore the format of the executable varies depending on the platform targeted. Windows Portable Executable is the primary format for Windows family operating systems for example, while Mach-O is the dominant format for MacOS operating systems. These executable formats cannot be considered file formats of the same kind as data file formats (for example, MP3 audio files or PDF documents). The latter type is generated to conform to a file format specification that allows them to be decoded, independently of platform, by software. Software programs are written for decoding by hardware (usually via an operating system), and

contained within a file structure which ensures they can be loaded by the specific target platform.

Symbolic and executable representations can be considered on a spectrum of human readability, which I term *opacity*. Source code is relatively transparent: this code is intended to be written by humans and so it is possible to read the code and interpret what it does.[5] Compiled software on the other hand, is relatively opaque. Its inputs and outputs are usually apparent, but the actual mechanisms of the software—the sequences of low-level operations such manipulation of data or arithmetic calculations—are hidden from view, rendering the software (for practical purposes) a black box. Even if the machine code expression of these mechanisms were to be examined, its interpretation would be impossible for those without specialist knowledge, and time-consuming for those with. For most users of software, what the program is doing is hidden beneath the surface—be that behind screen output or some other manifest behaviour. Where there is no transparent representation of the software program available, then, it is likely to be challenging to work out what that software is doing, and as a result to document and debug it. Finding ways to manage opacity therefore becomes an important consideration when working with software—a topic I return to later in this chapter.

2.2.3. Abstraction and the Materiality of Software

While the opacity problem introduced in the previous section can make understanding software difficult, its cause is fundamental to the way in which humans interact with computer programs: detail is hidden so that the user can focus on what is relevant in the given context, through a process known as abstraction (Guttag, 2013). Programmers code in, and compile from, high-level programming languages (as opposed to machine code or the closely linked assembly language) so that they can focus on writing a program to achieve a goal without including the large amount of instructional detail required to carry out basic operations. Email clients present a button that a user can engage to send an email, rather than have them deal directly with the appropriate email protocols. While the practical benefits of such uses are clear, the downside to the prevalence of abstraction is that in most cases those who

[5] There can be variance in the degree to which source materials are transparent, particularly in relation to whether the code has human authored comments, the programming approach taken and whether the full code base is available. This is an important issue in relation to software documentation which I discuss in depth in Chapter 5.

engage with software (including those that create it) are to some extent removed from the concrete realities of the computational processes that underlie their operation. Thoroughly addressing the technical characteristics of software therefore involves grappling with how any one view on the software might be presenting abstractions from technical detail.

Nick Montfort proposed the t*erm screen essentialism* to describe the contrast between contemporary readings of textual works of new media (such as interactive fiction), which are focused on screen outputs, and the early days of computing in which interaction with computers was largely paper based (Montfort, 2005). Montfort argues that the reader's connection to the "formal workings" of such programs has been lost through a focus on screen outputs. The punch cards of early computer systems for example, bore a clear physical signifier of their connection to the stored information: the holes themselves. The digital document as rendered by a document reader, on the other hand, bears little resemblance to its underlying representation as code. The "screens" of screen essentialism are not always involved in the use of software, yet a similar phenomenon can still be observed; the perceptible traces of formal workings are lost in the artifice of the software's manifestation. I propose *experiential essentialism* as a more general term for this phenomenon, which encompasses any tangible action of a software-based artwork. The primary use of the concept for this research, is to contrast engaging with a software-based artwork as a tangible phenomenon with the deeper (and typically more technical) level of engagement required of the conservator.

These issues relate closely to notions of software's *materiality*—that is, the significance of its basis in physical substance. As something which might be considered intangible, is it possible for software to possess a materiality? Matthew Kirschenbaum, building on the ideas of Montfort, proposes a two-part conception of materiality for digital media: *formal* and *forensic* (Kirschenbaum, 2012). Formal materiality, much like Montfort's "formal workings", concerns an interrogation of the digital object and its environment, below the screen itself but at a level still removed from any physical trace. This offers an extension of the concept of materiality beyond physical substance, to also encompass computational abstractions, such as the interface of an operating system or the textured surface of a 3D object. Formal materiality is further complicated when applied to software due to the status of the software super-object as manifold: each sub-component of the super-object presents different formal qualities and could be considered materially distinct. Forensic

materiality on the other hand, concerns the potential for uniqueness among all digital things, if they are examined to a low enough level of detail. All computational phenomena are ultimately rooted in a physical substrate of some kind, be that the magnetisations of hard disk platter or the charged capacitors of a random-access memory chip. These are potentially “individualizing” physical traces in the study of digital artefacts (Kirschenbaum, 2012).

Rigorous study of software as a material then, necessitates understanding the levels of abstraction at which it is and has been engaged—by both creator and user (or viewer)—and how they interact. I will end this section by presenting a model of the levels of abstraction at which we might need to address software in a conservation or digital preservation scenario. While preceding the work of Montfort and Kirschenbaum, Kenneth Thibodeau proposed a tripartite model for understanding the different levels of interaction we have with digital objects (Thibodeau, 2002) that retains relevance in light of their conclusions. Thibodeau suggests that digital objects can be understood as having properties addressable at three different levels: physical, logical and conceptual. These levels are described in Table 2 below, accompanied by my own examples of how their principles might be applied to software.

Digital object level	**Definition (adapted from Thibodeau, 2002)**	**As applied to software**
Physical	The object as an inscription of signs on a physical medium.	The physical representation of the software on a physical substrate, such as the sequential patterns of magnetisation on the surface of a hard disk drive platter to represent the binary bits that make up the software program.
Logical	The object that is recognised and processed by software.	The symbolic representation of the software that is machine actionable, such as executable machine code, compilable source code, or a technical interface made available to another software system.
Conceptual	The object as it is recognised and understood by a person.	The software as a system of inputs and outputs which can be perceived by an agent (usually a human), such as the modulated light emitted by an LCD display or the response of a set of files to a

		drag-and-drop action via an input device.

Table 2. Representation of Kenneth Thibodeau's properties model for digital objects, with original examples provided to demonstrate how its principles might be applied to software.

I propose that these three levels are one way in which we can understand an expanded notion of the materiality of software; one that helps us in appropriately addressing questions relating to the conservation of software-based art. Sometimes the appropriate level at which to work in order to answer a question relating to a software program might be clear. Storage concerns and maintaining the bit-level integrity of files would be addressed primarily at the physical level. Connectivity to another software system would be understood through technical interfaces addressable at the logical level. In other cases, however, there will be a need to navigate connections between the levels. The experience of pressing a button on a website, for example, is one which is tactile at the conceptual level, while also providing computational instruction at the logical level which might flow into bit-level change at the physical level. The qualities of a rendered image are understood by a viewer primarily on the conceptual level, yet their formation requires addressing the processes occurring at the logical level. Understanding software holistically, therefore, requires operating at the boundaries between different materialities.

2.3. Software Performance Model

While the physical and logical layers of the model introduced in the previous section are persistent, the conceptual layer is ephemeral: when software is not being executed, it is impossible to address its conceptual properties directly. As described earlier in the chapter, execution is the point at which latent software becomes actualised, and the host computer system begins to process and act upon the encoded instructions for as long as they specify or until the process is terminated. This process yields the manifest behaviours of the software and in turn, the tangible characteristics of a software-based artwork.

The ephemeral nature of this process and the instructional nature of the code invite analogies to *performance*. This is not a new idea in the study of digital media: Lev Manovich introduces a similar notion in *Software Takes Command* (Manovich, 2013). Using the phrase "software performances" (p.33), Manovich emphasises how the "media experience constructed by software usually does not correspond to any single static document stored in some media" (p.34) but rather is subject to the design of the software it is viewed with. While Manovich's focus here is on software as a means

of rendering other files, the focus of this research is on software (or rather, the software super-object) as the source of the performance in and of itself. As a result, the focus shifts from the experience constructed by software in relation to a data object upon which it acts, to one constructed by the software itself and its host execution environment (itself composed of software and hardware). In this case then, the host computer system is the performer and the software its instructions. To further extend the analogy of theatrical performance: the performance involves more than just an actor (the execution environment) and a script (the code)—it also involves props (data sources). The form of these data sources may be various, ranging from resources packaged with a software program (e.g. graphics assets used in a user interface) or external services (e.g. geolocation data fetched via a web API).

This idea of software as performance relates closely to Clifford Lynch's formalisation of "experiential" digital objects, which emphasises a shift in the focus of digital preservation "from the bits that constitute the digital object to the behaviour of the rendering system" (Lynch, 2000, p.36-37). A formal model to describe these kinds of performance was initially created by the National Archives of Australia (NAA) (Heslop, et al., 2002) in the context of digital records. This model specifies a sequence of events: source, process and performance. Within the NAA model, a *source* is a "fixed message that interacts with technology" (p.8) and must be combined with technology for it be of meaning to a user. The *process* is "the technology required to render meaning from the source" (p.8-9). Together, the source and process combine to create a *performance,* which a *user* (a person or machine) is then able view. The experiential qualities of a digital record are therefore essentially ephemeral, and its qualities contingent on the hardware and software processes involved in its performance. As later demonstrated during the InSPECT research project, this model can provide a framework for understanding how the properties of digital objects are not inherent, but rather the result of a process of interpretation and rendering (Knight, 2009).

The NAA model was first applied to software by Matthews, Shaon, Bicarregui and Jones (Matthews, et al., 2010). This usage adapted the model to apply to "software products", a term which the authors use to refer to programs designed for the playback and processing of data—thus framing software as a means of creating of a "data performance". While software programs (including all the case study artworks I am examining) typically do involve an element of data processing, the structural and conceptual relationship between software and data may vary considerably.

Furthermore, for software-based artworks, software is the medium of creative expression in and of itself—not simply a tool for rendering. Therefore, the original NAA model seems a more suitable starting point for a model of software performances that can be applied to software-based artworks.

An adaptation of the NAA model is presented in Figure 2 below. In this version of the model, the “Researcher” element has been removed and bounding boxes have been added to delineate two distinct phases to the performance: source and process exist within an execution environment, whereas the performance occurs externally of this in a performance environment. While this model is relatively simple and makes concessions regarding the actual complexity of its elements (for example, the source element may be made up many interdependent components), it provides a base on which to build within this thesis.

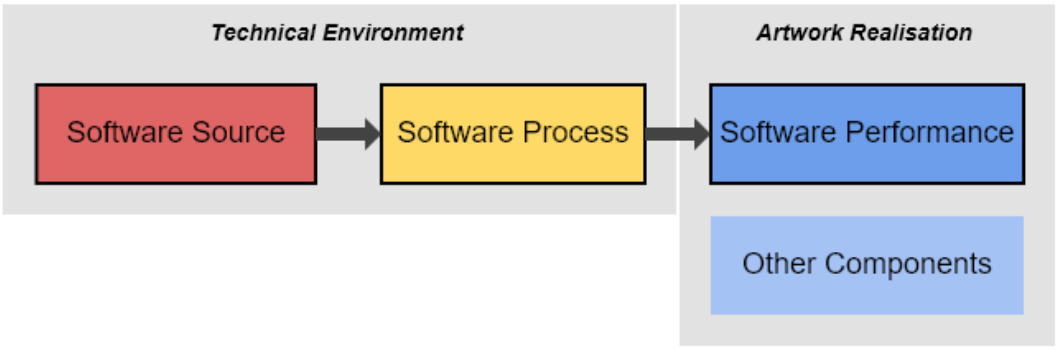


Figure 2. Visual representation of the software performance model, adapted from the National Archives of Australia’s (NAA) performance model for digital records. Coloured boxes indicate the components of the model, while grey boxes indicate the environment within which they exist or occur.

The most immediate practical implication of this software performance model is that we must consider whether each execution of a software program has the potential to be different. While the instructional nature of software might imply that there is limited room for interpretation, there are two reasons that the results of computational processes might vary. The first is simply that the instructions themselves may introduce randomness to the performance, through for example, an algorithm that creates probabilistic behaviour. The second is that, while the process might follow the precise logic of the instructions, it can only act within the capabilities of the hardware and software environment in which it executes. The power of a computer system’s hardware for example, might result in the rendered output of a program being generated at a visibly slower or faster rate. Therefore, the precise components of the

system and their configuration may also have a significant impact on the nature of the performance and therefore the human experience of software.

In the context of this research, the software performance model must also be considered in relation to how the artwork as a whole is experienced—software may only be a part of the works material constituents. Pip Laurenson argues that time-based media artworks should be considered “installed events”, realised as part of a two-stage process (Laurenson, 2006). Laurenson builds her argument using the theories of Nelson Goodman, who set out a distinction between autographic and allographic artistic works in *Languages of Art* (Goodman, 1968). Autographic works are one-stage works, such as a painting, wherein their replication does not result in an authentic realisation of the work—it could only be considered a forgery). Allographic works are two-stage, such as a musical composition, the authenticity of which resides in its score—thus requiring enacting (with potential degrees of variation) each time it is realised. Any performance of an allographic work can be considered essentially authentic. Laurenson uses this distinction to explain how works which are realised in two phases, such as time-based media artworks, demand careful consideration of acceptable parameters of change between installations (Laurenson, 2006).

In a paper presenting an approach to the documentation of time-based media installations developed at the Guggenheim, and building on Laurenson’s theory, Joanna Phillips points out a persistent terminological confusion over the label for an occurrence of a time-based media artwork in conservation literature (Phillips, 2007). The terms Phillips highlights include “manifestation,” “realization,” “materialization,” “representation,” and “instance”. It is useful to consider Brian Castriota’s crystallisation of the type-token distinction as the forbear of the autographic-allographic divide: an occurrence of an artwork is a *token* to the artwork’s *type* (Castriota, 2017). There is a common philosophical basis in related terminology then, the broad applicability of which might explain its proliferation. *Realisation* is given preference within this thesis as it emphasises the processual nature of tokenisation, while maintaining a link with the phrasing used by Laurenson—other similar terms are largely avoided.[6]

[6] “Manifestation” and “materialisation” (the latter of these is rather infrequently used in the literature) are not preferred as this language implies a physicality to the token. “Instantiation” is also avoided, as it has distinct meanings in computer science (particularly object-oriented

The significance of this term in relation to the software performance can be further clarified using concepts from formal ontology (Spear, 2006). A realisation is an occurrent entity, in that it has temporal parts and exists only partially at any given point in time. Most of the components of a realisation, on the other hand, are continuant entities, in that they have no temporal parts and are persistent through time while maintaining their identity. However, while a software super-object or a particular projector might be examples of continuant entities, the software performance itself, much like the realisation, is an occurrent entity. If we see both software and the artwork itself as essentially temporal and performative, how do these two levels of performance relate to each other? Laurenson briefly considers this relationship between the media itself (with an emphasis on moving image) and the larger realisation of the artwork. She concludes this by stating that:

> "An element of indeterminacy is central to the idea of a work being performed, and this indeterminacy is not present in the playback of media but is present in the act of installing an installation." (Laurenson, 2006, para. 28)

While there are degrees of difference in the level of indeterminacy, recent research suggests that contrary to this assertion, playback of media such as a digital video does in fact have an element of indeterminacy as a result of contingency on the features of the playback system used (Rice, 2015). Can software be said to have a similar (or analogous) contingency? Evidence from research by Agathe Jarczyk into the emulation of Cory Arcangel's *Super Mario Clouds,* indicates that the visual output of the software employed by Arcangel has a level of contingency on its execution environment—in this case different NES console emulations give slightly different results (Jarczyk, 2015). Whether this might have wider applicability (particularly outside of an emulation context) is unclear from existing research. In the next section, I will argue that the key to addressing this may lie in the relationship between software and the technical environment in which it is executed.

2.4. Software and Environment

programming) and information science (in the construction of representations of knowledge). "Iteration", Phillips choice for the Guggenheim documentation model, is contextually useful, but implies lineage through time and progressive change which may not always be applicable. Finally, "representation" does not suitably describe an artwork performance, as any individual token could be considered authentic rather than representational.

In this section I examine the composition and boundaries of the software super-object, particularly its close relationship with the technical environment in which it is executed. There has been a tendency to characterise software-based artworks as *complex digital objects* in digital preservation research (e.g. Enge, & Lurk, 2014; Konstantelos, et al., 2012, Rechert, et al., 2013). Given that "digital object" is generally defined as one or more bit sequences (CCSDS, 2012, anon. The InterPARES 2 Project Dictionary, 2014), the portion of the software super-object which resides in digital files would certainly seem to fit within this definition. However, the ideas of the software performance and potential indeterminacy introduced in the previous section indicate that the situation may be more complex than this. This is important for purposes of preservation because we want to be able to identify how a particular software performance is achieved and perhaps reproduce that software performance.

Compiled software programs or binaries[7] were introduced earlier in this chapter as the executable representation of a software program, as opposed to a non-executable representation such as source code. The basis of a software performance is often more complex than a single computer program however, and I am using the term software super-object to describe the set of binaries and associated data which form the source components of the software performance model introduced in the previous section. More concretely, the software super-object can typically be understood as comprising digital files structured and linked in some way which is meaningful to the host system when the software is executed. In some cases locating these resources at time of execution (or *runtime*) may involve using operating system functionality (system path calls) while in others it may use the location of the binaries as a relative point from which to traverse the file system. While directories are not a meaningful indication of the reality of file storage media (directory systems are an abstraction of these structures to enable easier file management), it is sometimes helpful to focus on the collection of files contained within a directory as the object of preservation. A compressed bundle (e.g. a ZIP or tarball file) of this application directory is a common way to distribute software over the internet, while software installers contain instructions to establish appropriate directories and any necessary references in a location in the host system. As I will go on to demonstrate, limiting engagement to this level is unsuitable for effectively managing software-based artworks in the long-

[7] Another related term is 'application software', which is used in contrast to 'system software' to distinguish user-oriented and support-oriented software. The distinction is difficult to apply to software-based artworks and their complex ontologies, so is avoided in this thesis.

term. This is because the process element of the software performance model transcends the software super-object that it takes as source, by employing the constellation of interconnected components that make up the *technical environment* in which the process is launched.

Determining the extent to which a software super-object can be moved between technical environments while maintaining the characteristics of a software performance involves many variables. These variables can be understood in relation to the components that form the technical environment, and their individual configuration. In examining the software employed in the case study artworks examined within this thesis, we can identify certain recurring types of components that make up these environments. These components are visualised in Figure 3 and described in further detail below.

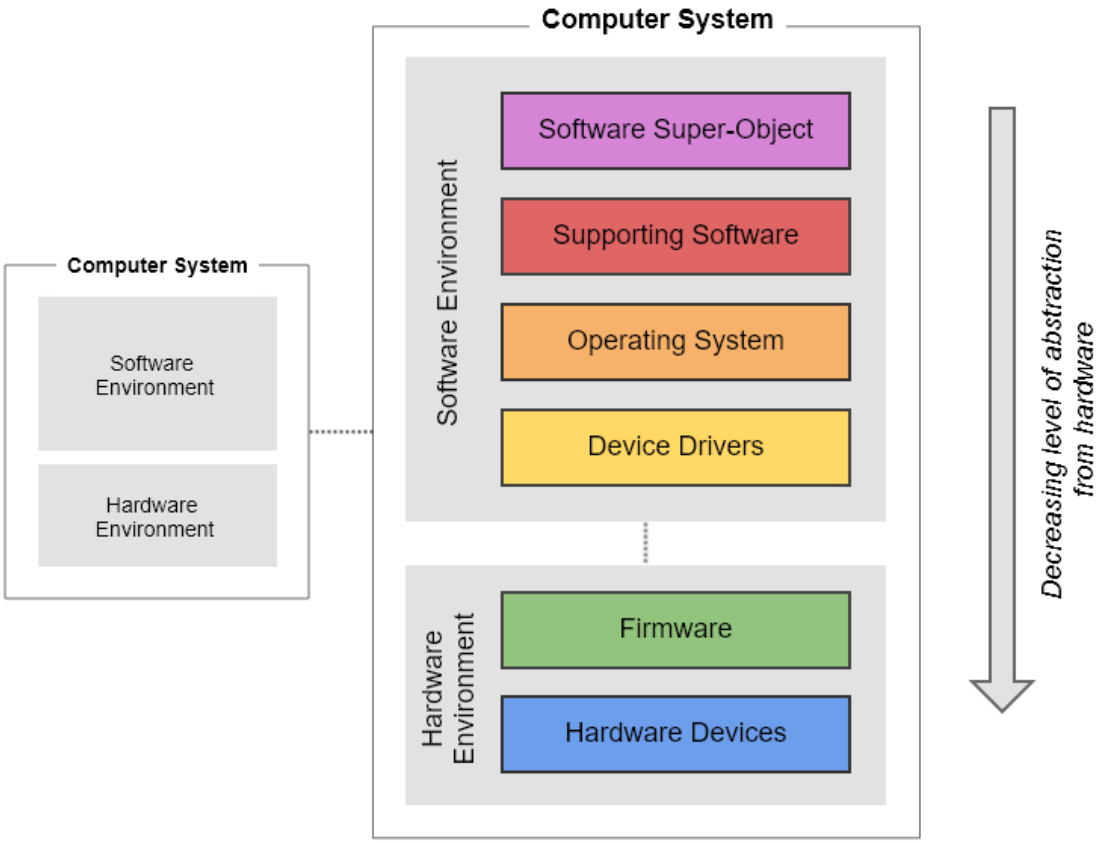


Figure 3. Representation of the generic structural components of a technical environment consisting of two linked computer systems (the smaller computer system is simplified for clarity, but would also contain components). Coloured bounded boxes indicate component layer types (description can be found in the main text), while grey unbounded boxes indicate environment types. Dotted lines indicate technical interfaces between environments.

As illustrated in the diagram, the component layers that make up a technical environment can be divided into those forming software and hardware environments respectively. Data exchange can occur between any component and that of another computer system, provided a suitable interface is available. The software environment is composed of four possible component types:

- **Software Super-Object:** A subset of software consisting of binaries and data assets which perform some function or purpose. This component is a simplification of what may be a very variable structure.
- **Supporting Software:** Other software components which support the function of the software super-object, including (but not limited to) runtime libraries, runtime environments, APIs and databases, where these are not considered part of the operating system.
- **Operating System:** A specialised form of software supporting the execution of software programs and communication with hardware and other components. An operating system is usually composed of a kernel—the primary control system—and supporting interfaces, frameworks and services.
- **Device Drivers:** A specialised form of software which supports communication between software, operating system and hardware.

In practice, some of these layers may mesh very closely. Some device drivers, for example, are a core part of an operating system to enable access to generic hardware.

The hardware environment has two components types:

- **Firmware:** A form of specialised software which is stored in hardware and provides core functionality. Despite firmware being software, its inextricable link to hardware means it should be considered part of the hardware environment. Firmware may not always be present on hardware.
- **Hardware Components:** The physical components that provide individual functionality and when assembled make up a computer system.

Technical interfaces exist between hardware and software environments. These interfaces separate the software super-object from the specifics of hardware to some degree, meaning that in many cases a specific hardware component can be swapped

for other similar hardware without affecting the software performance. Furthermore, hardware may be emulated using software, at which point it becomes a part of the software environment. In practice this means that in many cases making exchanges in the hardware component layer may have relatively little impact on the software performance, providing sufficiently generic hardware has been used (Rechert, et al., 2016). The more likely areas of influence on the software performance result from connections between the software super-object and other components in the software environment. Connections of this kind can be referred to as *dependency* relationships.

The term dependency is sometimes used in a programming context to describe the internal relationships within program code, but is here used to refer strictly to external dependency relationships. This also excludes dependency relationships between digital objects within the software super-object, which are links established within the code. Digital preservation researcher Klaus Rechert has developed a typology for dependencies based on their relationship with the software program component of a software-based artwork (Rechert, et al., 2016). There are two crucial distinctions identified. The first is between abstract and specific dependencies. The former would only require the presence of a non-specific component to provide generic functionality (e.g. any graphics API capable of rendering 2D graphics), while the latter would depend on a specific component (e.g. the OpenGL API). The second distinction is between direct and indirect dependencies. The former describes dependency relationships posed by the software program (i.e. the target of preservation efforts), while the latter describes the possibility that a component linked by a direct dependency may itself have dependency relationships with other components. Indirect dependencies may result in an exponential increase in technical environment complexity, as any one linked component may itself pose multiple dependencies. Dependency relationships may therefore form complex networks, which might be understood as a graph representing the directed relationships between components.

The reproducibility of software environments is further complicated by the configuration of the individual components that are linked in the dependency graph. Any component within a computer system may have parameters or settings which can be changed between performances, including both the software super-object and the components on which it poses dependencies. In the former case, configuration might be managed within the application directory through text files or as variables stored within the executable itself. John Gerrard's *Sow Farm*, for instance, employs

a 32-bit Windows Portable Executable file containing the program code and data, associated with a set of plain text files which contain variables loaded by the software program on execution. These can be altered by opening the text files in a text editor and changing the values. In the case of Rafael Lozano-Hemmer's *Subtitled Public* software, the configuration files are managed behind the scenes, a Graphical User Interface (GUI) providing a user friendly interface through which to edit them. Configuration may also be associated with environment components on which the software depends, introducing further challenges to understanding the parameters of a particular software performance. Different component types and versions may present variable configuration processes and option sets, and require careful examination to fully understand and document.

In practice, the extent to which performance indeterminacy is caused by variations in technical environment appears to be variable. For example, the 2010 realisation of Michael Craig-Martin's *Becoming* employs a software super-object consisting of a single Windows Portable Executable file. This file encapsulates everything required to execute the software performance correctly, providing it is hosted within a range of suitable Windows operating systems (OS) and connected to appropriate display hardware (an LCD screen in a custom case). Suitable Windows OSs range from Windows XP (released in 2001) to Windows 10 (released in 2015). Providing the hardware provides simple graphics rendering capabilities and sufficient processing power, an accurate software performance could be achieved within a variety of technical environments. John Gerrard's *Sow Farm* on the other hand, involves a set of interlinked files including a Windows Portable Executable, data assets, libraries and text configuration files. This software super-object requires not only a Windows operating system between versions Vista and 10, but also a set of other software components that are correctly configured and installed, for the desired software performance to be achieved. I return to issues of performance reconstruction and verification later in this thesis.

The software super-object, the digital materials at the heart of a software-based artwork, have an extremely close relationship with their technical environment. Not only might a very specific component need to be present within this technical environment, but it may also pose its own dependency relationships and require configuration in a certain way to generate the desired software performance. The ability to reconstitute this performance is not only desirable in the immediate examination and display of a software-based artwork, but also for ensuring that the

parameters of a software performance are understood so that they can be maintained in future performances where desirable. As a result, the activity of identifying and understanding the various elements of the technical environment has significant value for the long-term preservation of the software-based artwork.

2.5. Emergence of Software as Medium

So far within this chapter, software has been considered with the aim of characterising its properties as a material, and largely in isolation of its use by artists. The first issue to address, is the need for a baseline understanding of how to distinguish between medium and material, concepts which I introduced at the beginning of this chapter. Our understanding of *material* has been further clarified earlier in this chapter in relation to various notions of materiality. For software, material describes not only the substance of software in a literal sense (understood as its forensic or physical materiality), but can also be used to describe the logical and conceptual layers of software (understood as its formal materiality). In developing a clearer notion of *medium*, we can look to the philosophy of art. Philosopher David Davies has developed a theoretical framework of medium in art which helps us clarify the concept, defining the artwork as “an artistic statement as articulated in an artistic medium realized in a vehicle” (Davies, 2004, p.60). The *vehicular* medium is the substrate or substance of the artwork (which might range from a physical object to an action carried out by a performer), whereas the *artistic* medium is the means through which the artist imbues the artwork with meaning, through their intentional manipulation of the vehicle. Crucially for this discussion, the artistic medium need not constitute an element of the works realisation.

This distinction allows further refinement of the definition of software-based art: software-based art is that where software materials can be seen to constitute both the primary vehicular medium *and* primary artistic medium. Considering some hypothetical examples provides a demonstration of how this might be applied. A software generated image, ink-jet printed on paper, could be considered an example of the use of software as an artistic medium. However, as the artwork manifests as ink on paper, software could not be considered to constitute the vehicular medium of this work (which is ink and paper). An installation artwork which employed software to control lighting changes could be considered an example of the use of software as a vehicular medium. However, software could not be considered an artistic medium in this case, as the artist is not articulating an artistic statement through software—rather, the artistic medium is light, and software serves a purpose as a tool. However,

a superficially similar installation artwork which employed software to control lighting changes—but in the case based on data gathered live from the internet—*could* be considered software-based. Here software is not only a part of works realisation, but is essential to its understanding as an artwork.

As these examples imply, we cannot make the assumption that all artists use software in the same way, nor that the history of software as a medium will not challenge the various models I have presented to assist in the characterisation of software as a digital object. There has been considerable scholarly interest in the historical relationship between art and computers in the past two decades (Brown, et al., 2008, Shanken, 2009, Taylor, 2014 and Paul, 2015 represent a sample of this work.) However these histories tend to only give limited attention to those forms of use where software is *both* the artistic and vehicular medium of choice. It is not the aim of this section to attempt a treatment of this topic. Rather, I aim to simply identify some of the key software-based art related threads from the larger story of the relationship between art and technology, and through this develop a clearer picture of the significance of software as an artistic medium within both historical artistic practice and that of today. This process also offers a means of introducing and contextualising some of the significant genre terminology of relevance to the study of any kind of art with a significant technological component.

2.5.1. Computer Art and Historical Precedents

While the creative use of computer technology has occurred since the birth of modern computational paradigms in the 1950s, the rejection of art of this kind by critics and the commercial art world at the time of its creation (Taylor, 2014) has resulted in a patchy historical record. Renewed interest in the 21st century has seen parts of this history emerge, through projects such as the CACHe project, which culminated in an edited volume on the subject (Brown, et al., 2008), and the work of others such as Christiane Paul (Paul, 2003), Edward A. Shanken (Shanken, 2009) and Grant Taylor (Taylor, 2014). When examined in relation to their coverage of software and computation, these historical accounts focus largely on what is termed *computer art*. This is a broad term that can encompass any kind of art which involves a computer in its production or display and seems to far precede use of the word ‘software’ to describe artworks which involve software. As a result, the computer art canon includes many examples of works which we would consider software-based. There are important distinctions between these terms however, which I will clarify below through the use of two historical examples.

Among the earliest examples of the use of a computer to create something framed as art were Ben Laposky's *Oscillon* series, the first of which was produced in 1952 (Victoria and Albert Museum, 2011). These works involved the use of analogue computer equipment to produce abstract forms which were displayed on the screen of a cathode ray oscilloscope, which would then be captured using long exposure photography and printed on paper for the purposes of exhibition (Laposky, 1969). An example, *Oscillon 19*, is reproduced in Figure 4 below.

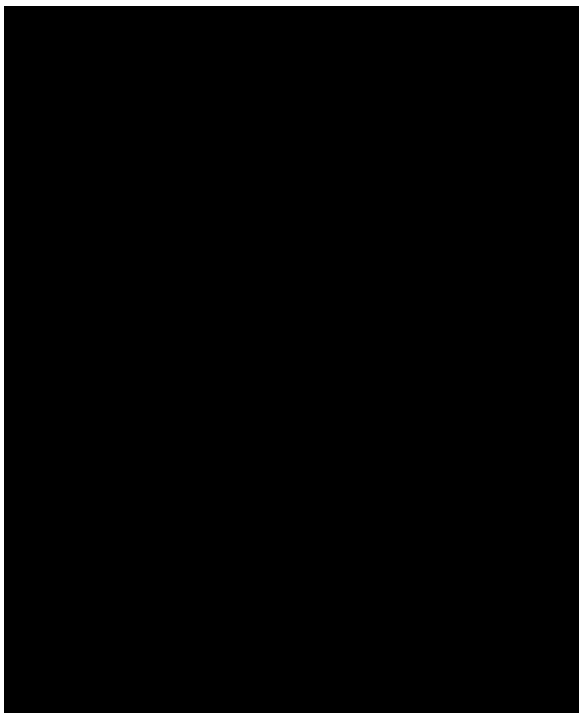


Figure 4. Reproduction of *Oscillion 19* (1952) by Ben Laposky, from *Oscillon: Electronic Abstractions* (Laposky, 1969). © Ben Laposky and MIT Press.

Nick Lambert points to Laposky's *Oscillons* as a pivotal moment in the emergence of computer art and other forms of technology-based art, as for the first time art was created outside of the constraints of a physical medium (Lambert, 2003). The screen outputs certainly had many of the process-driven and iterative characteristics of software, but Laposky's electronic manipulations did not involve software in a strict sense—that is, encoded instructions were not processed by a computer system. The oscilloscope was controllable via a physical interface of knobs and buttons, so in theory it would have been possible to recreate particular configurations—but this bears little resemblance to code-based computer programs. It does however, potentially align with the software performance model introduced in the previous section of this chapter: a data source (wave generator) is being realised as a performance (the CRT output) by a process (which occurs within the oscilloscope).

The additional step of photography complicates the nature of the performance, however, as it is unclear whether Laposky viewed the printed photographs as the primary artistic output, rather than the actual shapes displayed on the CRT. The use of photography may have been a practical concession to allow the works to be displayed independently of the technology, or it may have been viewed by Laposky as an essential step in the works' realisation. This ambiguity draws attention to an important distinction between software-based art and computer art: the former must always be executed in software at the time of exhibition, while the latter may refer to artworks where there is a transition from a software medium to paper (or another non-software medium). Understood in relation to the artistic-vehicular distinction developed earlier in this chapter, computer artworks often employ software only in the sense of artistic medium. Software's presence as a constitutive part of the artwork when it is realised is extremely important in relation to conservation: the conservation of an *Oscillon* printed on paper, for example, would demand a different set of considerations from the conservation of the means to produce them. However, in both cases the technological means of production would remain of great conceptual significance, given the level of interest Laposky expressed in his writings (Laposky, 1969). We can conclude, therefore, that the use of software as an artistic medium is not necessarily indicative of its interest in relation to the goals of this research.

The potential for software be constitutive of an artwork is often contingent on it having a storable form—thus allowing repeat performances of the encoded instructions[8]. The storage of a computer program in electronic memory (essentially the foundation of what we understand as software today) was first achieved in 1948 by a team at the University of Manchester using their Mark 1 computer (Lavington, 1998). While there is limited information about the patterns of creative experimentation involving software that followed, some of the earliest exhibited examples were the cybernetic sculptures of Nicolas Schöffer. The earliest of these was *CYSP 1* which was first exhibited in 1956 (Dreher, 2014). This work used a computer—sometimes referred to as an "electronic brain" (Dreher, 2014)—developed by the Philips Company[9] to

[8] It should be noted that truly ephemeral software programs (e.g. self-destructive) could be employed by artists, although the author is not aware of any examples.

[9] Artist collaborations with commercial and military groups were frequently the means by which art and technology could cross-pollinate during the early days of computer art, due to the high cost and limited availability of computers at the time.

process and convert light and sound inputs into the movement of the parts (including wheels on its base) of a kinetic sculpture (Hoggett, 2017). While there is limited record of the technical components of the work available, it seems likely that the computer used by Philips would have contained stored routines or algorithms. Shanken's account of *CYSP 1* supports this conclusion, stating that it was "programmed to respond electronically to its environment, actively involving the viewer in the temporal experience of the work" (Shanken, 2002).

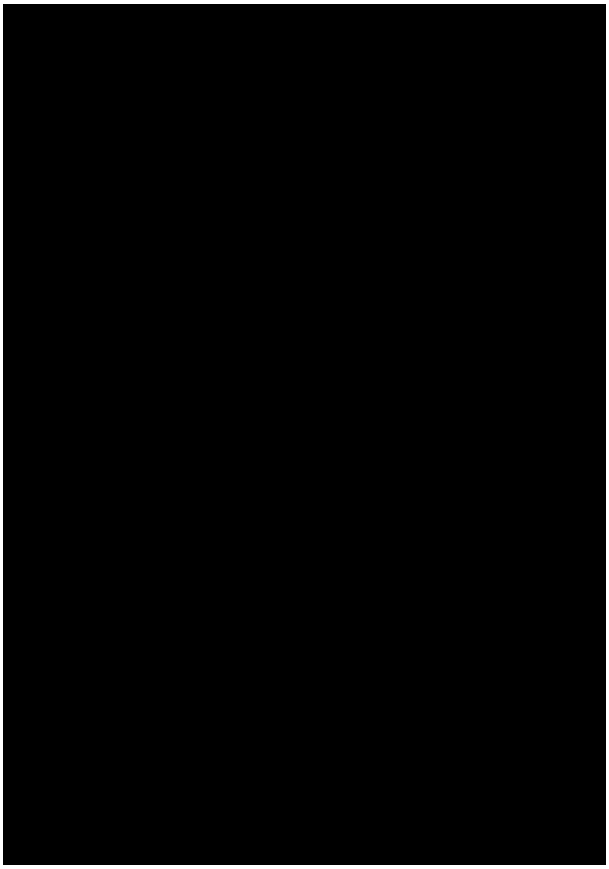


Figure 5. Photograph of *CYSP 1* (1956) by Nicolas Schöffer. The movements of the sculptural array at its top and wheels at its bottom were controlled by a computer concealed within the black cylindrical base. © Nicolas Schöffer and Reuben Hogget.

The work also responded to people in its proximity and was intended to be shown with dancing performers. *CYSP 1* and similar artworks that followed in the 1960s and 70s bear a remarkable resemblance to software-based art as we understand it today, particularly in their use of interactivity. They embody the liveness and performativity of software within such works, which can exist only in their fully realised form while code is being executed.

Aside from those historical developments which we can isolate due to the involvement

of identifiable technology, during a similar time frame other forms of art were being developed which have revealing similarities. Shanken has proposed that the parallel emergence of conceptual art and what he terms “art-and-technology” is associated with the transition into the Information Age (Shanken, 2002)—that is, a shift in focus among many economies from traditional industry to information technology. Lev Manovich earlier proposed that these two parallel worlds—he refers to them as Duchamp-land (the art world) and Turing-land (the computer art world) respectively—have fundamentally different outlooks and are unlikely to ever converge (Manovich, 1996). However, Shanken highlights how a number of individuals were moving fluidly between the two camps: Jack Burnham curated the 1970 exhibition *Software* at the Jewish Museum in New York, which juxtaposed works of conceptual art with displays of technology; while artists such as Roy Ascott and Hans Haacke have found favour on both sides of the divide. The connection between these two worlds is important, as it invites consideration of the extent to which software can be considered as a medium outside of the technological frameworks of its definition. Florian Cramer describes an algorithm-like, instructional form of poetry (the process he describes involves using coin flips to generate a new poem from an existing poem) as essentially akin to software (Cramer, 2002)—though in this case they are forms which would be theoretically executable by either human and machine. Returning to the software performance model introduced in the previous section, we find that it could also be applied to understanding Cramer’s poem program: a set of instructions (source) are used to generate a poem (process) resulting in an audible rendition (performance). The implication of this is that theory which can help us understand the conservation of conceptual and instruction-based art, might also help us understand the conservation of software-based art—an idea I return to later in this chapter.

2.5.2. New Media and the Computational Metamedium

The commercial possibilities of software were being realised by the late 1960s and the first software companies began producing tools to aid the programmer (Haigh, 2011). The rise of the personal computer in the 1970s and 80s (Ceruzzi, 2003) saw increasing demand for software, the emergence of new programming languages and tools, and a host of new technologies. By the 1990s, an increasing diversity of computer-related technologies had caused the term computer art to begin to be replaced by a more nuanced lexicon which included internet art, interactive art, generative art and software art (Taylor, 2014). We also see the emergence of the term *new media* (and thus *new media art*) to describe the growing use and significance of these diverse technologies based in computation. The term new media

is itself lacking any widely agreed upon definition—indeed, Lev Manovich devotes an entire chapter to describing it in *The Language of New Media* without arriving at a succinct definition (Manovich, 2001). In the study of new media *art*, the situation is not much clearer. Mark Tribe and Reena Jana's treatise on the genre opens by defining new media artworks as those, "that make use of emerging media technologies and are concerned with the cultural, political, and aesthetic possibilities of these tools" (Tribe, & Jana, 2006, p.6). In an interview, curator Steven Sacks frames new media art more as a way of thinking than an identifiable movement, suggesting that it is:

> "not just about being new—it's a contemporary way of thinking and responding to the latest tools of creation and societal changes. Each generation reveals their own 'new media art' based on current influences and the latest technologies." (Goldstein, 2014)

Despite a lack of clarity over its definition, the term new media art remains in use and software-based artworks often fall within its broad umbrella. Indeed, the proliferation of new media and the parallel development of accessible software programming gave rise to new kinds of software, which further complicate our understanding of software as medium. I will consider two of these new forms below.

The first is the emergence of software to generate art which mimics non-computational media—for example, email mimics letter-writing while digital painting tools mimic traditional painting processes. In Manovich's *Software Takes Command*, he uses the term *cultural software* to describe software which enables cultural activities relating to creativity and communication (Manovich, 2013). He traces the origin of cultural software back to research by Alan Kay and Adele Goldberg at Xerox PARC in the mid-1970s (Kay, & Goldberg, 1977), in which the authors offer a vision of computing where the computer is more than just a tool of business and industry, but a tool for creativity (Wardrip-Fruin, & Montfort, 2003). They suggest that the computer could provide "a metamedium, whose content would be a wide range of already-existing and not-yet-invented media" (Kay, & Goldberg, 1977, p.40). To illustrate this they offer a number of prescient proposals for the use of such a metamedium, including an architect being able to simulate 3D space during the design process, and a composer having the option to easily edit and listen to their score as they wrote it. Cultural software such as CAD and audio sequencing tools were later developed to fulfil these roles, with the computer as the interface with this metamedium.

An implication of the metamedium concept is that software might not be considered a distinct medium at all when considered at the conceptual level—rather, due to the compound nature of the software super-object and the potential to engage with it at various levels of abstraction, it is one which can present multiple materialities. Some of these materialities might relate to pre-computational media. As Nick Lambert notes, these replications of existing media "are not judged by standards derived from their computational origins, so much as the visual and experiential connections with older media" (Lambert, 2010, p.89). A software program which generates a moving image and is viewed as a projection, for example, might be considered in relation to the language of cinema. The extent to which the projected image might be considered in relation to the languages of *new* media would relate to how conspicuous the medium is made. If the moving images had signifiers of 3D graphics (for example, visible texture tiling or aliasing artefacts), they might be considered in relation to video games, for example. The actual means of expression—code or production software—is not always signified at the conceptual layer. This points to a feature that distinguishes software performances from other kinds of performance: the precise mechanism of the performance is typically not visible. In theory, this might allows for potential changes in the source element of the performance model, without impacting the integrity of the performance providing its characteristics are maintained.

With the rise of cultural software, so too came an increased distance between the artist and code (Taylor, 2014). While early computer artists had to grapple with the technology using a limited range of languages and hardware, increasing availability of programming and production tools would begin to see the underlying technological frameworks obscured. This shift in working practice may not have altered the significance of code as material, but it certainly affected its significance as medium. This curious relationship between artist and code was explored by curator Christiane Paul in the online CODeDOc exhibitions, one for the Whitney Museum of American Art's Artport in 2002 (Paul, 2002) and a second for the Ars Electronic Festival in 2003 (Paul, 2003). For the CODeDOc exhibitions, source code was presented alongside the artwork it generated, inviting the contemplation of code as both mechanistic and aesthetic consideration. In all of the six case studies I examined, the behavioural qualities of the artworks are a product of a degree of programming; in only one did the artist have direct engagement with the code itself. This factor may not alter the significance of code at the logical layer (it remains at least a historical artefact), but it certainly would have an effect on its conceptual significance. The relationship between code, as an individual expression, and the artwork is a topic of importance

which I return to later within this thesis.

The second significant new form of software to emerge with new media was that associated with the internet. Artists engagements with these technologies resulted in a new genre known as internet art (or net art) (Greene, 2004, Paul, 2015). The term internet art is generally used to refer to art which is made for dissemination over the internet, although its usage varies as with other genre terms introduced here. Because of the networked means of accessing such works, this is a practice that is geographically diffuse, and that has responded quickly to technological developments. It is important to note for the purposes of this thesis that much internet art can also be considered software-based art, as most works involve both remote (e.g. web servers, databases, APIs) and local (e.g. web browsers and their plugins) software programs. The connectivity in this sense is crucial to their understanding, and therefore poses a significant challenge to the repeatability of the performance model. Internet art also posed challenges to the mainstream art world's ability to collect its art. Indeed, its collection and exhibition in conventional art spaces has caused considerable debate amongst those who contributed to its history. In 1997 internet art was included in documenta X (David, 1997), a first for the documenta series—a major event in the mainstream art world calendar. The inclusion was controversial among artists, with the selected artworks being consigned to their own room which was visually themed to feel something like an office space filled with desks and desktop computers. Artist duo Jodi (whose work was included in the exhibition) called the internet art room an "unnecessary, confusing symbolic construct", which they felt artificially grouped artists whose only similarity was their shared choice of media (Jodi, 1997). Showing a sensitivity to the context in which internet art—and indeed, other forms of software-based art which might be experienced outside a typical gallery setting—is likely to be an important consideration in their restaging and long-term preservation.

While the use of software within art has continued apace since the events of the 1990s, there has been a gradual process of integration into artistic practice which marks a shift in focus from media-centric exhibiting to one in which the use of technology is informed by a set of cultural conditions rather than as an end in itself (Wiley, et al., 2013). While this shift was signposted by exhibitions such as 010101 at SFMOMA in 2001, which exhibited both new media and traditional media artworks side-by-side (Graham, & Cook, 2010), it has only more recently become widely acknowledged. This shift is reflected in the appearance of terms such as "post-

internet" (Olson, 2012) and "neomateriality" (Paul, 2015)—both of which suggest an environment in which the digital is becoming more firmly integrated with existing languages of art. With this shift has come the increased attention given to software-based art as something of conservation concern. As technology continues to evolve, new challenges may emerge rapidly. There is an opportunity for the conservator, therefore, to take a crucial role in connecting the evolving artistic metamedium of software with the material concerns it presents.

2.6. Medium-Specific Conservation Considerations: A Lexicon

In this chapter I have explored a range of issues relating to the technical characteristics of software and its status as a medium and material of artistic expression. As a preliminary advancement in the development of this conceptual framework, we can revisit the working definition of software-based art provided in Chapter 1. We can now clearly define software-based art is that for which software is the primary artistic medium *and is executed at the time of the work's realisation.* To conclude this chapter, I will use the knowledge gathered to build a lexicon of terms to describe the medium-specific conservation considerations presented by software-based art. These considerations are not necessarily unique to software-based art but are connected within it in such a way that they find new meaning. The six key concepts that form the lexicon are: performativity, functionality, structural complexity, opacity, liminal materiality and multiplicity.

The idea of **performativity** reflects the fact that the realisation of a software-based artwork is to some degree ephemeral—it is contingent on the continued activity of a process running on a computer system. This can be formalised using a model of software performance: a *source* consisting of executable code (perhaps linked to other digital resources) is executed as a computational *process* (or processes), yielding a *performance* (i.e. the experiential elements of the work). Understanding this is important because if software performances are to be reliably recreated (a requirement of long-term preservation), there is a need to manage any potential for variability within the form and interpretation of the executable code. While in Section 2.3 I highlighted evidence to suggest that differences in execution environment may introduce variability into a performance, this area remains relatively unexplored territory for software-based art. If we are to understand the software-based art conservator's role as one which centres on achieving consistent software performances through time, there is a need for new approaches to identifying and documenting acceptable parameters of change at the software level. Addressing this

gap is one of the major goals of this thesis.

Other kinds of digital art might also be considered performed in a similar sense—a quality which relates to the presence of software within all digital environments. As Christiane Paul has pointed out in relation to difficulties in defining software art, “every form of digital art employs code and algorithms at some level” (Paul, 2015, p.124). Digital images require rendering while digital video requires playback—both of which require software. However viewing software *itself* as the source of a performance (as it is software-based art), rather than as the mediator of a performance (i.e. a media playback mechanism) presents different considerations. This is because unlike other forms of digital media, like a digital video file which contains a set number of frames to played back in chronological order, software is instructional: the host computer acts upon encoded instructions to achieve some result. This has been characterised in various ways by other authors: Steve Dietz calls it “computability” (Dietz, 2000), while Pip Laurenson frames it as software-based art’s capacity to “do something in real time, something more than playback, so that the input is different from the output” (Laurenson, 2013, p.77). These qualities might be understood as relating to the inherent **functionality** of software—all software is created to achieve an effect of some kind. In a conservation context, it is important to understand this functionality, because if it is possible to identify and express it, it is then possible to understand what the software’s purpose within the artwork is and how it might be maintained.

The potential for functionality resides within the software super-object, a compound digital object that may be comprised of numerous interconnected components linked by code. This **structural complexity** presents itself in a variety of ways. At its most basic level, software is itself not necessarily composed of a single discrete file, but rather a set of interlinked parts including additional executable code and data resources. The software may then also be inextricably linked to a certain execution environment consisting of particular software or hardware components (which can perhaps be configured in a variety of ways) on which it depends for successful execution. A number of authors have also identified the potential for software-based artworks to be “diffuse” (Fino-Radin, 2011, Laurenson, 2013)—that is, the software employed has connections to and dependencies on external systems and resources.

This has also been identified in relation to the networked properties of some software-based artworks, particularly internet art (Beryl, & Cook, 2010, Dekker, 2014). Such links may need to be maintained if the software is to be correctly performed, so changes occurring in these external resources and the means through which they are

accessed pose considerable risks in terms of long-term preservation. Furthermore, tracing connections may yield further connections—the output of one system can become the input for another (Dietz, & Altshuler, 2014), while dependencies can themselves have other dependencies. This potential for structural complexity may pose challenges in a conservation context because it makes understanding the complete software super-object harder, and because the maintenance of technical interfaces between components may be compromised by technological change. Thus achieving a reproducible software performance may become increasingly challenging. As a baseline, the relevant structures must be well understood to enable the management of this problem.

The effectiveness of the kinds of analysis required is likely to be further inhibited by other characteristics of software. In Section 2.2.2 I introduced the idea that software presents a variable level of **opacity**. Compiled software is essentially a black box system when it is running and can typically only be understood as a set of inputs and outputs. This means that the underlying code governing the behaviour of the software is largely hidden from view (as compiled machine code)—despite the fact that this hidden layer might be the one at which the artist is making important decisions (Dietz, & Altshuler, 2014). Examining this code is essential in order to elucidate the functionality that a software program has and ensure that a particular software performance can be repeated in the future. Therefore, opacity presents a significant conservation risk. This may be particularly significant where a human-readable representation of the software program (such as source code) is not available to consult.

The possibility of more than one representation or version of a software program can be understood as its potential for **multiplicity**. Software-based art is intrinsically multi-representational in that compiled binaries (a representation for interpretation by a computer system) are derived from source materials (a representation for human authoring and eventual compilation). Software-based artworks may also be modified and a new version of any the components of the software super-object generated, perhaps in the creation a new version for exhibition or to fix a bug. This is important simply because it is necessary to know what is being preserved, where it came from, and how it relates to the future realisation of the artwork— issues which are particularly critical when operating in digital environments which enable copying, transmission and the proliferation of digital objects. There is a need for some consistent means of structuring descriptions of versions, representations and the

relationships between them when writing documentation.

In addressing the technical components of a software-based artwork in relation to questions of meaning, we are confronted with a multi-faceted materiality which defies simple categorisation. As I demonstrated earlier in this chapter, the software super-object simultaneously presents several distinct material levels: the physical object (signs stored on a physical medium), the logical object (the symbolic representation of the physical object which can be executed) and the conceptual object (the manifest results of the processing of the logical object). Software presents a **liminal materiality**—that is, it simultaneously occupies multiple material states (which present different qualities and characteristics), without definitively belonging to any of them. As a metamedium (i.e a medium capable of reproducing other media) it has the potential to continue to evolve and so present new material qualities, as illustrated by historical shifts in the range of technical possibilities available to and then utilised by artists.

Addressing this liminal materiality requires working outside of modes of experiential essentialism, and addressing underlying structures. While on a physical level software must be considered in relation to the physical characteristics of its storage, its tangible manifestations are ultimately meaningless without understanding them in relation to more abstract conditions of the logical layer—the decoding of signs, the rendering of pixels and the manipulation of interfaces. Navigating these various levels—their boundaries and connections—is the only way by which the conservation of software-based artworks can be meaningfully addressed. The individual significance of these levels in relation to a particular software-based artwork may vary considerably, and requires careful interpretation by the conservator. While to some extent we can understand the weighting of material concerns as defined by the artist's intentions and the work's production, it may also be modulated by expectations regarding the viewer's experience of the work. Graham and Cook suggest that in some cases a "viewer will 'see' this material [the visible manifestation] for the work and only with further investigation discover the layer of the work that is about the system, the flow, the interaction" (Graham, & Cook, 2010, p.62).

Navigating the subjectivity of viewer experience and the complex relationship between a work's tangible elements, its technical characteristics and its meaning, will be essential in understanding the identity of a software-based artwork and guiding efforts to preserve it. While software-based art remains a useful catch-all, relating to a specific challenge at this moment in time, as Rebecca Gordon points out in relation

to the phenomenon of expanded material range in contemporary art, “even when the same materials are adopted by different artists, a unifying interpretation of these materials is unlikely” (Gordon, 2013, p.8). In practice, we may be dealing with software-based artworks that present very different characteristics.

The six terms that form the lexicon of medium-specific conservation considerations described in this section can be summarised as follows:

- **Peformativity:** Software is experienced by the viewer as the tangible effect of instructions being executed by a computer system, which means that there may be potential for variation when this performance is repeated in a different environment.

- **Functionality:** In contrast to other digital media such as video, software is not played back—rather, it specifies instructions to achieve some effect. This means that in theory, there might be multiple ways to achieve this effect.

- **Structural complexity:** Software is not typically a discrete digital object, but rather presents a complex structure that includes linkages with its environment, including external systems and resources. This introduces difficulty in the restaging of software performances when this environment changes.

- **Opacity:** Different representations of software can be understood as falling somewhere on an opacity spectrum—the more opaque they are, the harder it is to understand how they work.

- **Multiplicity:** Software might exist in multiple representations, while copies and versions of a particular program might proliferate. This creates challenges in terms of the management of these different instances, particularly in maintaining their provenance and the relationships between them.

- **Liminal materiality:** Software has a curious material status that can only be understood by addressing it as if it possessed multiple materialities simultaneously. Understanding the significance of these different levels and the connections between them on a technical level will be important in addressing the conservation of a software-based artwork.

The terms introduced in this lexicon provide terminology for describing a set of key issues in developing approaches to the long-term care of software-based artworks.

Each of them will need to be addressed in any comprehensive framework for the documentation of software-based art. With this refined understanding of the software medium and its implications for conservation, in the next chapter I will consider the suitability of existing approaches for the documentation of software-based artworks.

CHAPTER 3

CONSERVATION DOCUMENTATION IN THEORY AND PRACTICE

3.1. Chapter Outline

In Chapter 1, I identified the potentially multifarious nature of documentation and a need to better understand how it might serve the conservation of software-based art. Equipped now with the prerequisite knowledge—a more complete understanding of the software medium as developed in Chapter 2—the purpose of this chapter is to consider existing conservation documentation standards, methods and approaches, and ascertain their suitability for the documentation of software-based art. At the end of the chapter, I will have arrived at some conclusions regarding the areas requiring most research attention, which will serve to guide the structure of this research and the following sections of this thesis.

Grappling with the nature of the document was a prominent concern of the early pioneers of what we now know as the field of information science. In the first part of this chapter I revisit historical documentation theory in relation to the technological changes of the past few decades—the characteristics of software in particular—with the aim of more clearly delimiting the scope of the document within this research. In

the second part of the chapter, I consider the practical implications of this theoretical framework in relation to museum practice, through an examination of the role and activities of the conservator. I look at the core components of the conservation workflow, including the documentation approaches employed, and assess whether they might be applied to the documentation of software-based artworks as-is or where new methodologies may need to be developed.

3.2. Revisiting Documentation Theory

The origins of the term *documentation* are shared with those of *document*, and can be traced to the latin *documentum*, meaning lesson, proof, or written evidence (Duranti, & Franks, 2015). While these origins are still to some extent evident in the use of the word today, documentation might now be used to refer to a nebulous array of materials that extends far beyond. For insight into the development of contemporary notions of the document, we look to a group of European pioneers (based mostly in libraries) who were known collectively as the "documentalists" (Rayward, 1996). This group of thinkers, active from the early to mid- 20th Century, set out the foundations for our understanding of documentation today by redefining what a document could be. Prior to their work, the term documentation was almost solely used to refer to the management of documents for scholarly use—documents being effectively limited to printed texts (Buckland, 1997). The documentalists, beginning with Paul Otlet's *Traité de documentation* in 1934, began to develop an expanded understanding of document to include, for example, museum objects and explanatory models.

Several decades after Otlet, Suzanne Briet developed these ideas further in her 1951 treatise, *Qu'est-ce que la documentation?* (*"What is documentation?"* in English). This text contains a definition of document that remains impressively representative of our multi-faceted understanding of the word in information science today. The definition, this version taken from a recent translation of the original French text, posits the document as:

> "any concrete or symbolic indexical sign [indice], preserved or recorded toward the ends of representing, of reconstituting, or of proving a physical or intellectual phenomenon" (Briet, 2006, p.10)

Briet's decision to refer to the object of documentation as sign or "indice" has positioned this definition favourably for the later development of digital documents and computational paradigms such as the semantic web, as well as other

unconventional documentation types. Indeed, this definition allows for a broad variety of materials to be considered documentation. Briet provides the famous example of an antelope: a specimen of the animal, she suggests, becomes documentation when captured and entered into a museum collection.

The three “ends” to documentation specified by Briet all have significance in the context of conservation. “Representation” was introduced in Section 1.2.2 in relation to both software and documentation. In a documentation context it relates to the potential for a document to depict or act in place of something else, an important and broadly relevant concept in conservation documentation and one I discuss in more detail later in this chapter. “Reconstitution” is highly significant in time-based media conservation, where conservators might be interested in documentation that supports the future realisation of a work, whether that work is specified as a specific set of components or with more flexibility. Finally, “proof” relates closely to notions of evidence and authenticity. Documentation might provide substantiation of authenticity in a direct way, such as an artist-signed certificate of ownership or an artists approval of some conservation action. Importantly for conservation documentation however, the notion of proof links to the value that any document attempting representation or supporting reconstitution might have. Evidence of authenticity in documentation is how we understand it to be reliable or trustworthy.

In the same text, Briet outlines some of the potential forms documentation can take. Of particular interest in our further refining the limits of documentation, is a breakdown of these forms according to the concepts that documentation can “make known”. While the complex, performative nature of time-based media art is not easily reconciled with Briet’s now dated examples, this structure still provides a helpful lens through which to gauge the problem space. Based on the knowledge gathered in Chapter 2, we might consider software-based artworks to span three of Briet’s suggested targets (or “objects”) of documentation, existing simultaneously as concepts (or ideas), artistic creations, and events (or activities)—and therefore not classifiable within the same framework. Using Briet’s principles, I have developed a typology relating to the time-based media art domain, illustrated with contemporary examples of real-world documentation practice. It should be noted that these types are non-discrete, and any single document may belong to multiple categories—rather than offering a taxonomy, these categories serve to highlight the range of things which can be considered documentation in this domain.

1. Documentation can be **descriptive information** about an entity or event

(e.g. an exhibition catalogue text for an artwork; a description of the components in an installation).

2. Documentation can be an **abstract representation** of an entity or event (e.g. a diagrammatic representation of an installation; an artwork metadata record).

3. Documentation can be a **concrete representation** of an entity or event (e.g. a scale model of an installation; a photograph of an installation).

4. Documentation can be a **token representation** of an entity or event (e.g. a sample of data produced by a generative artwork).

5. Documentation can be a **surrogate representation** of an entity or event (e.g. a scale model used for planning; a simulation model used for testing).

6. Documentation can be a **resolvable reference** to an entity or event (e.g. a collection number or identifier; a citation).

7. Documentation can be a **reproduction** of other documentation (e.g. a quotation; a photocopy).

8. Documentation can be a **description of documentation** (e.g. a metadata schema; a standard).

As I will go on to demonstrate in this chapter, all of these types of documentation might find use in the conservation of software-based artworks. While representing a diversity of very different forms, what all of the types have in common is that they must all be created with reference to an entity or event of some kind (Briet's "physical or intellectual phenomenon"): the object of documentation. This is sometimes called *indexicality*, referring to the document's semiotic function in acting as an index or pointer (Day, 2016). This is important within the understanding of the document concept as applied to this research, as without their indexicality documents lose their meaning. The findings of Chapter 2 suggest the software-based artworks may present a particular challenge to indexicality. While a software performance could be considered an event, it is an event associated with the coming together of a certain constellation of components. These components, such as the software itself, have porous boundaries and may have multiple forms and versions, making the network of references between document and object potentially expansive. A complete treatment of issues regarding consistent identification of digital resources is beyond

the scope of this research, but is revisited in the context of documenting artwork life histories in Chapter 6.

3.2.1. Representation, Modelling and Use

The indexicality relationship bears no greater weight than where documentation is representational, as it is in this role that the document must be able to act in place of the object of documentation. In this section I will take a brief aside to consider the significance of representation in conservation documentation, particularly in the creation of highly structured documentation such as diagrams, metadata and ontologies. Challenges around creating effective structured representations can be considered in relation to *modelling:* the process of creating *models*. A model, I here define as a representation of a system for some purpose—usually informational, interrogative or analytical—and to some degree possessing the ability to stand in for the thing it represents. For example, a model of a climate system might be used to forecast weather, and as such stands in for the climate system so that the forecaster does not have to deal with the much higher levels of complexity the real climate system presents. The origins of modelling are in the physical sciences and formalisation of scientific theory, but since the emergence of computing, the practice of constructing models has been applied as an experimental method in the humanities (Schreibman, et al., 2004, Terras, 2005, Ciula, & Eide, 2014). The development of any system of documentation involves some degree of modelling, whether that be in the elements to be drawn in a diagram or the metadata elements to include in an information architecture. In practice, a model of some kind (even if not explicitly referred to as such) typically forms the theoretical basis of documentation templates, frameworks and methodologies.

One of the major challenges in modelling is how to ensure a model's utility as a representation, where it is constructed for some purpose. Difficulty arises in deciding what to model, a problem well illustrated by the Jorge Luis Borges' parable (presented with fictitious accreditation), *On Exactitude in Science*:

> "...In that Empire, the Art of Cartography attained such Perfection that the map of a single Province occupied the entirety of a City, and the map of the Empire, the entirety of a Province. In time, those Unconscionable Maps no longer satisfied, and the Cartographers Guilds struck a Map of the Empire whose size was that of the Empire, and which coincided point for point with it. The following Generations, who were not so fond of the Study of Cartography as their Forebears had been, saw that that vast Map was Useless, and not without some Pitilessness was it, that they

> delivered it up to the Inclemencies of Sun and Winters. In the Deserts of the West, still today, there are Tattered Ruins of that Map, inhabited by Animals and Beggars; in all the Land there is no other Relic of the Disciplines of Geography.
>
> —Suarez Miranda,Viajes devarones prudentes, Libro IV,Cap. XLV, Lerida, 1658"
>
> (Borges, 1999)

The Empire's impractical map alludes to one of the key tensions in the construction of any kind of model or knowledge representation system: a balance of accuracy or completeness against usability. This tension is known as the map-territory relation. In this case, the accuracy of the map has been given priority over the usability of the map, thus rendering it useless. While the absurdity of Borges' story serves an illustrative purpose, real world examples of balancing usability and accuracy in representations might be much more nuanced. How then, would we assess whether a representation is successful and so avoid creating our own "Unconscionable Maps"? There is clearly a need for abstraction of complex systems, but the extent to which abstraction can or should be made without compromising their value is less clear. There is little literature exploring this topic in the domains of art conservation and digital preservation documentation. However, richer theoretical discussion of representation can be found within political science and scientific simulation.

In political science this discussion relates to the potential ability of a candidate or government to represent their people. Despite this very different context, this domain is relevant to this discussion as it also pertains to representation *in place of* another thing, much as structured representations of a thing act in place of the thing they represent—whether that be for information retrieval or some explanatory purpose. In the 1960s, Hanna Pitkin developed a classification of representation types through an examination of the word's use and the differing meanings which emerge (Pitkin, 1967). Pitkin's types, Dovi suggests, could be used as a standard for assessing a representative (Dovi, & Zalta, 2017). Summarised in general terms, the types present a set of criteria:

- **Formalistic representation:** the level to which the representation is able to act in place of the represented;
- **Symbolic representation:** the significance of the representation for the represented;
- **Descriptive representation:** the extent to which the representation

resembles the represented;

- **Substantive representation:** the use which the representation receives in service of the represented;

These criteria highlight a number of important characteristics of representation in the context of cultural artefacts, and form a useful set of baseline criteria for assessing a representation's value. For three of the four criteria, there appears to be no upper bound on the extent to which that type of representation would be desirable: the more formalistically capable, symbolically significant (this could be seen as relating to ideas of authenticity) and descriptively accurate the representation is, the more successful the representation would be. In many cases, availability of information may place a limit on the extent to which these criteria can be met, but in a hypothetical situation where all information were available, the problem of the map-territory relation is encountered: we have simply created a replica of the represented.

It is the fourth criterion—substantive representation—which may provide the key to managing the map-territory relation by placing a requirement of *use* on the representation. The value of descriptive metadata, for example, would be judged not only by its success at descriptive representation of the work in question, but also by its use value in conveying appropriate information succinctly to someone browsing a collections database. Digital preservation metadata on the other hand, might be judged by its success as a formalistic representation: that is, it must be able to be acted upon in place of the digital object itself. However, the extent of the actual information required is governed by the types of preservation process which might be applied to the object by a preservation system—so defining the use value of this representation. Later in this chapter I return to ideas of representation in relation to use, and explore how a use criterion can be used to interrogate existing approaches to structured representation in relation to the conservation of software-based art.

3.2.2. Information Science and Digital Documents

By the 1990s the documentalist tradition was considered a part of the broader discipline of information science, which Saracevic defines as "the science and practice dealing with the effective collection, storage, retrieval and use of information" (Saracevic, 2017, p.1). It has been suggested that information science should be considered a kind of meta-discipline through its shared borders with the many other disciplines that must also navigate these issues (Bawden, & Robinson, 2012). Despite it being less recognisable as a distinct field of practice, there was renewed interest in

documentation science in the mid-1990s, triggered in part by a need to revisit old questions in light of the growth of information systems and new forms of digital document (Levy, 1994, Buckland, 1997). The work of Briet, Otlet and the documentalists was revisited at this time, with scholars finding that their theories of documentation—as functional and framed by use, rather than form—helped provide a meaningful lens on this new, dematerialised document (Buckland, 1997).

New approaches to documentation theory that have emerged since then have often focused on the social construction of the meaning of documents and identifying the forces that shape their creation (Levy, 1994, Buckland, 1997, Zhang, & Benjamin, 2007). While the technology employed is identified as a common means of understanding documentation, Levy emphasises that documents are ultimately social artefacts: they must be understood with respect to their use (which Levy identifies more specifically as "work"), particularly in relation to the human activities and institutions within which they are embedded (Levy, 1994). This helps deal with the impractically broad documentalist conception of the document as almost anything, by allowing us to define documents through the human creation or designation of a document. Sabine Roux crystallises Jean Meyriat's distinction between these two types as "'documents by intention,' which are produced from the start with the aim of communicating, and 'documents by attribution,' which become documents when the user uses them to search for information" (Roux, 2016, p.4).[10]

Armed with this theory, we can begin to answer questions about the limits of what can be considered a document within this research. An important preliminary question is whether or not the artwork can or should be considered a document in and of itself. In her study of the documentation of internet art, Annet Dekker argues (building on documentation theory) that such works might indeed be considered documents when their properties are examined (Dekker, 2014). Looking at this idea in relation to Meyriat's distinction, it is clear that software-based artworks are not documents by intention: an artwork is the product of an artistic intention, not a documentary one. If the artistic intention is also documentary (for example, it employs photographs which document a subject), then it may be said to have documentary properties, but it remains essentially an artwork. Nor can software-based artworks be understood as

[10] This quotation contains phrases translated by Roux from Meyriat's 1981 article *Document, documentation, documentologie*, for which a general English translation is not yet available and so could not be consulted directly.

documents by attribution: they do not typically hold a use value in relation to information retrieval. It is possible that in the future artworks might become of interest to historians of the time as, for example, a proxy for their social conditions—therefore conveying information to some degree and so gaining documentation-like properties. However, there is a lack of a clear indexical relationship with an object of documentation in such a scenario, and I therefore reject the idea that artworks should be given document status.

While I propose that artworks and documents are distinct concepts (at least for the purposes of this research), it is important to note that the components that make up an artwork may become documentary when they are not resolved or resolvable into a realisation of the work. Traces of the artwork may be particularly distinctive in some cases. For example, if the work involves a prominent sculptural component this may act as an effective signifier of the nature of the original installation, even outside of its original context. This is an important theoretical issue for museums engaged with the care of ephemeral works, as it gives value to the components of a work even where further realisations impossible. Within this research however, I will primarily focus on those documents which are authored with an *intention* of documentation, as only this type can be meaningfully addressed using the constructive research methodology. Based on the typologies developed, the primary forms of documentation with which I expect to engage are *informational* (being designed to convey information) and *representational* (being designed to represent a thing). Given the assertion that documentation is created through intention to document, I propose that the problem of documenting software-based art may be addressed through the identification of the *purpose* of this documentation. In the next section I consider the purposes which may emerge in a museum conservation context. This permits a closer examination of the relationship between documentation, the needs of the collection or object in question (in this case software-based art) and the technological approaches available.

3.3. Documentation in the Conservation Workflow

In the preceding sections I identified that the types of documentation generated in a particular conservation context will usually be documents by intention—so derived from some purpose—and that they may serve informational and representational functions in relation to the thing they document. Ultimately a documentation purpose responds to some identified need and so a document's value will be understood through its actual use in serving this need. In this second part of the chapter, I will

consider the potential needs which conservators might have in terms of conservation documentation for software-based art and the extent to which these might be addressed using existing approaches.

The conservator's views on documentation might be best understood through the conservation workflow: the phases of action which make up the conservator's engagement with a particular artwork. This workflow is variable in its exact formulation and inherently non-linear, as works are revisited through time both as part of regular collection care procedures and for the purposes of a specific display of the work. Nonetheless, there are certain identifiable stages and activity areas which help us isolate the use that might be made of documentation in the service of conservation processes. I will look at each of these in turn in the following sections and examine how they might need to be reconsidered to accommodate software-based art. The structure of these sections has in part been shaped by my experiences within the Time-based Media Conservation team at Tate, but also by published methodologies and information gathered during interviews and research visits during this research (all of these sources are referenced within the text where specifically drawn upon). Given the potential relevance of this research to institutions or individuals with differing resources or interests—perhaps an artist or collector developing their own strategies—I have divided the text into modular sections and phrased them as generically as possible. It should be emphasised however, that there is no truly generic workflow for conservation; this is merely one perspective.

There have been several comparisons of the prominent models for the documentation of time-based media art (Jones, 2008, Heydenreich, 2011, Dekker, 2013). These are thorough examinations of the models they cover and reveal something of the considerable breadth of work in this area, but are flawed in that they attempt to compare models with very different purposes. Rather than use a similar comparative approach, my aim is to contextualise individual components of these models in relation to the area of practice to which they might apply, and in doing so reach more concrete conclusions regarding use value in relation to software-based art. In the following sections, I look at the workflow according to three activity areas: Acquisition, Ongoing Care, and Information Systems. For each I explore the applications and limitations of current approaches to documentation when applied to the unique conservation considerations posed by software-based artworks. Gaps are identified within each section of the text, and are then considered in terms of their implications for this research in the final section of this chapter.

3.3.1. Acquisition

Acquisition is a term used to describe the process through which an artwork is brought into a collection. While acquisition involves many parties within the museum and the clarification of issues outside of the scope of this thesis (such as ownership and copyright), it also represents the first steps in the documentation process for conservators. Some documentation guidelines suggest a more granular breakdown of acquisition into distinct sub-phases (e.g. pre-acquisition, the phase where the viability of an acquisition is explored before it is formally agreed), such the Matters in Media Art approach (Matters in Media Art, 2015). In practice, the goals of these sub-phases overlap considerably in relation to documentation considerations and can be characterised by an increasing level of detail as an acquisition gains momentum. I therefore identify the overarching goals here, rather than the incremental steps. Conservation processes involving documentation which occur at acquisition might include:

- Developing an understanding of what the artwork *is*, the artist's intentions in its making, and its significance as an addition to the collection.
- Developing an understanding of the work's technical components, including what will be acquired (computer systems, digital files etc.) and the basic parameters of installation or display.
- Carrying out initial consideration of risk for identified technological components and developing a plan for their long-term care, including consideration of costs.

The first of these two processes involves consultation and compilation of existing documentation, particularly that created by the artist and other parties involved in its creation, exhibition and care prior to acquisition. This might be characterised as information gathering. The last process and the formulation of a plan for the long-term care of the work, involves analysis of the information gathered as well as the artwork itself. This stage can be characterised as conservation planning and culminates in the formulation of a structured document which captures the plan developed. In the following three sections I look in turn at the documentation requirements of information gathering, examination of materials, and conservation planning.

3.3.1.1. Information Gathering

Documentation processes during the early stages of acquisition could be

characterised as driven by information gathering rather than analysis. For software-based artworks, the extent of the information gathered may require reconsideration in light of the significant differences between software (as explored in Chapter 2) and the media types which many existing guidelines have been developed to address. The primary aim of this section then, is to present a preliminary exploration of the targets of this kind of information gathering process. The information gathered at this stage is a significant factor in making informed decisions about the future of a work, particularly in assessing the viability of the acquisition, and later in developing an appropriate plan for the long-term care of the work in relation to risks of loss and obsolescence. I will begin by considering the kinds of existing documentation which might be gathered together at this stage.

The DOCAM (from the French project name, '*Documentation et conservation du patrimoine des arts mediatiques')* Documentation Model was developed during a Daniel Langlois Foundation project running from 2005 to 2010 (DOCAM, n.d.). This approach explicitly models the Creation stage of an artwork, which includes the conception and production of the work. Where the context is a museum environment, this Creation stage must be considered at time of acquisition as it will typically only be understood through documentation of the process acquired with the work. The risk of losing this documentation increases the more time has elapsed since creation, and therefore gathering documentation associated with the creation of a work should be a high priority during acquisition. While several time-based media documentation models offer typologies of documents which provide a starting point for information gathering (DOCAM, n.d., V2_Institute for the Unstable Media, 2004), these models make limited reference to the technical documentation of the software development process. To identify where these models might be expanded, the use of software engineering approaches has been explored by several authors (Marchese, 2011, Engel, & Hellar, 2014, Engel, & Wharton, 2014). Documentation is a significant component of software engineering practice, with well-established standards which aim to ensure that a software system can be effectively maintained in the long-term. Within this field, units of documentation are commonly referred to as *artefacts*[11], a broad term which can denote any "self-contained work result" of software engineering processes (Fernández et al., 2018, p.12) ranging from design materials to an actual

[11] In this thesis the British English spelling 'artefact' is used, but it should be noted that the US English 'artifact' is more common in software engineering literature.

software product.

The conservation community seems to have arrived at a consensus regarding the significance of one particular artefact of the creation of software: source code. Its value in the documentation and conservation of software-based art is now well established (Enge, & Lurk, 2013, Engel, & Wharton, 2014). This resonates with the results of surveys of documentation practice in software engineering (Lethbridge, et al., 2003, de Souza, et al., 2006). The value of source code stems from the fact that it represents what the program does in a human readable form. As discussed in Chapter 2, source code can be considered another representation of the low-level code that is contained within the executable software program: it essentially expresses the same set of instructions. However, acquiring source code may not be straightforward, as an artist may have never intended their source code to be shared or studied. Even where it is acquirable (and for all but one of the case study artworks examined, it was acquired), it may not provide the full picture. In reality, the creation of a software-based artwork can be characterised as comprising variable processes of programming and production, which may involve specialised development software and tools. Actually *writing* code may only be a part of the process. In the case of all six of the case study artworks examined within this thesis, development environments operating at various levels of abstraction have been employed in addition to the authoring of original code. Where access to or value of source code is compromised, there remains an open question as to whether the insights it reveals can be gained through other means—one I aim to address in this thesis (see Chapter 4).

Looking beyond source code, consensus on other important artefacts is less well established. Francis T. Marchese has suggested applying software engineering models to the documentation of software-based art. He proposes a set of generic and time-tested software engineering documentation artefacts (Marchese, 2011), which he later expands in relation to the Rational Unified Process model of software engineering (Marchese, 2013). Marchese's descriptions of these artefacts are reproduced below:

- "**Requirements** – Statements that identify the capabilities and characteristics of a digital artwork. This is the conceptual foundation for what has been created.

- **Architecture/Design** – An overview of software that includes the software's relationship to its environment and construction principles used in design of the software components. Typically a system's architecture is documented

as a collection of diagrams or charts that show its parts and their interconnections.

- **Technical** – Source code, algorithms, and interfaces are documented. Comments may be embedded within the system's source code and/or parts of external documentation.

- **End User** – Manuals are created (e.g., static documents, hypermedia, training videos, etc.) for the end-user, system administrators, and support staff.

- **Supplementary Materials** – Anything else related to the system. This includes: legal documents, design histories, interviews, scholarly books, installation plans, drawings, models, documentary videos, web sites, etc."

(from Marchese, 2011, p.305)

While Marchese's rigorous approach would likely be valuable in addressing conservation problems (as these established methods are for maintenance in commercial software environments), experience with the case study artworks examined for this research indicates that such a rigid formulation of documentation is unlikely to resonate with artists. Indeed, for the artwork case studies only end user (installation guidelines) and limited technical documentation (usually just commented source code) was supplied with the artwork. Furthermore, production of these artworks was a complex, often multi-party process and ultimately driven by the intention of creating art, not maintainable software. As Deena Engel put it in an interview in 2016, "I certainly wouldn't ask an artist to take time to do a UML diagram when they were busy creating art" (D. Engel, personal communication, 23 May 2016).

This is not to say artists do not think about technical documentation. A media artist's perspective on documentation is clearly represented in Rafael Lozano-Hemmer's "Best practices for conservation of media art from an artist's perspective" resource (Lozano-Hemmer, 2015). Lozano-Hemmer's suggestions for documentation are more loosely specified and include:

- **Working documents** such as "sketches, prototypes, parts lists, bits of research on the project".

- **Change tracking and versioning** of both code and other project documents

- **Bill of materials** list, which includes all the works components including

"brand and model, its function, the URL for information, and a small picture".

- **Read me** document to be bundled with software, including information about "operating system, DirectX, any graphics drivers, APIs, programming environments" required for its installation.

- **Artwork manual** which (incorporating some of the above) includes the follow parts: "i) a 'meta' narrative describing the key concepts and elements of the piece and how it works; ii) a detailed set-up procedure, including pictures of example installations, wiring diagrams, museographic notes such as desired lighting or acoustic conditions, sample layouts showing what is and is not allowed; iii) maintenance section on how to clean the piece and turn it on and off; iv) preservation section with the Bill of materials, all schematics, comments to the code."

  (Quoted text elements above are from Lozano-Hemmer, 2015)

In the same document Lozano-Hemmer suggests that artists might mistrust set conservation methods, which may not consider "the vast range of disparate experiences, methods, constraints and dependencies that can arise even within the work of a single artist" (Lozano-Hemmer, 2015). While likely somewhat tongue-in-cheek (after all, Lozano-Hemmer goes on to specify his own guidelines), this highlights the potential difficulties in a one-size-fits-all approach to conservation documentation and in placing any predefined expectation on an artist's working practice. Deena Engel and various collaborators have attempted to address this tension by exploring the generation of such documentation for works entering (or already in) museum collections either independently or in collaboration with artists. The document set explored overlaps with Marchese's, and includes source code documentation, high-level narratives describing code, flowcharts and UML diagrams (Engel, & Hellar, 2014, Engel, & Wharton, 2014, Engel, & Wharton, 2015). The construction of the latter three types is largely contingent on the former activity having been carried out, and so the potential applicability of these approaches is limited by the same difficulties I introduced in relation to source code earlier in this section. However, as the studies cited demonstrate, where it is possible to generate these in collaboration with the relevant expertise (and associated resources), their value is likely to be significant in the next steps of acquisition and conservation planning.

Using the recommendations of the studies discussed above, it is possible to develop

a generic, idealised classification of the documentation materials which might be acquired for software-based artworks. This classification, presented in Table 3 below, could be used as a prompt for conservators to identify and acquire these materials on acquisition. It should be noted that this table focuses on documentation types which are particularly important for software-based artworks, but does not include some of the more generic documentation types which might apply to time-based media art in general.

Document type	**Description**	**Example formats**
Source materials	The human-authored code and other production materials (including data assets) used in the creation of the software. Code should be acquired annotated with descriptive comments, or as a source code repository. Where relevant, this should also include important software and other production tools for accessing these source materials and potentially recompiling the software.	Code is usually stored as plain text, however if development software has been used the formats may be more complex, and have proprietary elements. Production tools may also be software. Data assets might be various e.g. SQL databases, images, video, 2D graphics, 3D models.
Installation documentation	Description of how the work has been configured and installed previously, including information about how it should behave and how it should look, and detailed instructions on how it might be reinstalled.	Documents, diagrams, screenshots/screencasts, photographs, videos, press materials.
Non-technical manuals	Manuals and other materials which describe the software and its use (usually in the context of display), usually for a non-expert user	Documents, screenshots/screencasts
Design documentation	Any design documentation that provides an overview of the software system such that its key components and their relationships can be understood, or the origins of the software in requirements, prototypes or other research.	Documents, diagrams.

Technical manuals	Detailed technical manuals for any off-the-shelf hardware or software components. For software this may include documentation of the development environments or programming languages used.	Documents, diagrams.

Table 3. Basic prompt list for the gathering of software-specific documentation at the acquisition of a software-based artwork.

The interview is a staple document of artist consultation in conservation practice and its nuances are well covered elsewhere (e.g. Crook, 2001, Beerkens, et al., 2012). However, I want to comment briefly on the potential impact of the qualities of software-based art on the interview process. Perhaps the most obvious, is the need for a specificity of questioning relating to the technical features of software as a medium. Based on the case studies examined in this thesis, artists are sometimes not involved in some of the lower level detail of the production of their software and often collaborate with specialists. As a result, there is a risk of information gaps—a risk heightened where there is a third party (for example, a gallery) involved in the artwork's custody transfer. This necessitates that the artist's *collaborators* be involved too where possible—a process which has been going on at Tate since their first software-based artwork was acquired in 2003. While production assistance is not uncommon in the production of contemporary art, the risk of lost knowledge is heightened where programmers are involved; this is because the understanding of the technical details may vary considerably between the two parties'. If a collaborator moves on, there is a high risk that that knowledge will be lost or become unavailable to the institution unless it is properly documented.

The principles of the interview might be further extended into documentation methods that aid this. One approach to this that particularly stands out among the Tate's existing body of documentation for their software-based artworks is something I will call the *walkthrough*. One party involved in the process is a non-programmer (typically a conservator), and the other is the artist, programmer or developer involved in the creation of the artwork. This person describes, in clear but technically accurate terminology, how the artwork functions and how it relates to the underlying software structures and the code. This description develops through a process of questioning driven by the conservator and provides a uniquely valuable insight through the lines of questioning which emerge, revealing information which the developer may not have

otherwise thought to convey. Equally, the conservator develops a much more nuanced understanding of the artwork's relationship with its programmatic basis and the decisions involved in its software implementation, than might be achieved through isolated technical analysis. In practice this document usually takes the form of a transcribed conversation or in some cases a chat logs from internet communication software. The walkthrough approach might be further extended by attaching the dialogue to a video screen capture of the digital resources in question.

3.3.1.2. Appraisal and Planning

The process of acquisition is typically evidenced by an initial report into the structure and condition of the artwork followed (assuming acquisition proceeds) by planning for its future care (Matters in Media Art, 2015). This initial assessment is based partly on the information resources discussed in the previous section, but potentially also by examining an artwork's components. The necessity for artwork analysis is intertwined with the act of documentation, a fact which is enshrined in the Code of Ethics and Guidelines for Practice created by the American Institute for Conservation of Historic and Artistic Works (AIC):

> "Before any intervention, the conservation professional should make a thorough examination of the cultural property and create appropriate records. These records and the reports derived from them must identify the cultural property and include the date of examination and the name of the examiner. They also should include, as appropriate, a description of structure, materials, condition, and pertinent history" (American Institute for Conservation of Historic and Artistic Works, 1994, p.9)

While the physical and hardware components will vary considerably, for software-based artworks analysis might typically focus on software in the form of digital files. Sometimes these might be delivered over the internet, in some cases on physical media (e.g. a hard drive or USB flash drive) or even as a whole computer system—all of these possibilities are evidenced among the artwork case studies I examined. Given the potential risks of acquiring digital materials on storage media, particularly that nearing or at obsolescence (Fino-Radin, 2011), acquiring digital files may be preferable. In either case, at this point of first contact, it becomes crucial that the integrity of the materials acquired is maintained by ensuring that the bits remain unchanged as they move between platforms and media: a concern known as "fixity" in the field of digital preservation (NDSA Infrastructure & Standards Working Groups, 2014).

Where storage media or a whole computer system is acquired, maintaining fixity requires the implementation of special procedures. When connecting storage media or powering on a computer system in order to extract examination copies of the digital files, there are risks of alterations to the data and file system. Furthermore, there is a requirement to know that the integrity of the bits has been maintained in any duplication procedures. The repurposing of approaches from the field of digital forensics has been found to help mitigate these risks. Digital forensics is a well-established field in law enforcement and security which, while seemingly far removed from the concerns of the arts, has been identified as an area with potential relevance to those working with digital cultural heritage (Kirschenbaum, et al., 2010, John, 2012, Dietrich, & Adelstein, 2015). The essential appeal of digital forensics in the context of acquiring digital artworks is that it provides a means to avoid risking alteration of data and to maintain fixity. This entails a strategy called write-blocking (usually using a hardware device that sits between the source and target), which prevents data write operations in the direction of the source.

A fundamental activity in digital forensics is combining write-blocking with principles of disk imaging (Woods, et al., 2011). Disk imaging can be used as a means to extract and encapsulate the complete data content of storage media for bit-level preservation (Rechert, et al., 2016). If this is done via write-blocking technology, a complete (mountable) representation of the data content has been acquired without any impact on the integrity of the original data. If the device imaged were originally bootable (typically if it was taken from a physical computer system), then this image can be used to reconstruct, via emulation or virtualisation, the original for purposes of examination and analysis. Even where this is not possible, it can act as a bit-for-bit backup of the original content of the drive. Working with disk images in this way poses many advantages over directly interacting with a computer system, including assurance of maintaining the integrity of the files in their original context, tracking (and reversing) changes made, and ease of manipulation and access.

While approaches such as disk imaging make accessing and analysing software more practical, methods for the actual analysis and documentation of results are still poorly developed. Existing templates for condition reporting, such as the Matters in Media Art Structure and Condition report for “computer-based” artworks (Matters in Media Art, 2015), provide little more than a prompt for information. As discussed in Chapter 2, the structurally complex and opaque nature of software-based artworks means that it might be difficult to identify the technologies used, while the functionality

and material complexity of the medium makes gaining this knowledge extremely important. Methods for the targeted analysis of software-based artworks are poorly understood however. The research outlined in the previous section has already highlighted source code analysis as a potential source of this information, yet this is time-consuming work currently being led by collaborations with computer scientists external to the museum (Engel, & Wharton, 2014, Dover, 2016). However, within existing research there is a lack of clarity over what the conservator might be expected to do and what might require collaboration with specialists in a particular programming language or technology. Given that the primary aim at acquisition is to work out what software is being used and how it might be installed and configured, spending resources on unearthing the details of technical implementation through source code analysis may be surplus to requirements.

At the confluence of many of the above concerns, is the aim of the conservation planning phase to identify those parts of the work which can be safely changed in order to achieve future realisations of the work. This relates back to the work of Pip Laurenson introduced in Section 2.3, who proposed that the *identity*[12] of a time-based media artwork can, to a varying extent, be detached from its constituent parts. This identity is understood through what Laurenson calls a “cluster of work-defining properties” that should remain consistent between realisations (Laurenson, 2006, para.50). In digital preservation an analogous concept known as *significant properties* emerged at around the same time, which digital preservation researcher Andrew Wilson defines as “the characteristics of digital objects that must be preserved over time in order to ensure the continued accessibility, usability, and meaning of the objects, and their capacity to be accepted as evidence of what they purport to record” (Wilson, 2007, p.8). These notions may help us manage change in a software-based artwork by establishing what is required to maintain an authentic performance of the work. Documenting work-defining or significant properties is likely to be crucial then, yet frameworks for doing so remain poorly developed. This is exacerbated by the fact that two levels of performance must be addressed: the realisation of the software-based artwork as a whole and the computational performance of the software super-object itself. In Section 2.3 I drew attention to research which suggests that consistent

[12] Where I use this term within this thesis, it can be understood as referring to what an artwork *is*. As I discuss later in this thesis, the idea of being able to pin down such a notion for an artwork is inherently challenging, but the term remains a useful concept from which to build this discussion.

playback of digital video requires careful management of the playback technology employed (Rice, 2015). For software-based artworks, there is a pressing need to identify whether any analogous concerns exist or whether software's functional nature overrides such concerns.

3.3.2. Ongoing Care

After the process of acquisition, an artwork formally enters the collection and impetus shifts towards completing documentation that is required for subsequent display and addressing concerns raised in conservation planning. Even where this sequential action is not apparent, the artwork is now in the care of the institution for the long term and becomes subject to monitoring and appropriate application of conservation strategies and treatments in the future. Taking Tate as an example, best practice for time-based media conservation has been to take what is referred to as an *active life* approach to preservation, which makes managing change a primary concern (Laurenson, 2015). Works are revisited "during the life of the artist, who may re-engage with the work at different points, but also beyond the life of the artist, as the work continues to be exhibited and displayed" (Laurenson, 2015). These revisits occur according to two rhythms: the first is that of the museum's collection care programme and an ongoing desire to display the work; the second is that of the medium, and so varies for works of different types. The regularity of significant change in software technologies (particularly in relation to patterns of software obsolescence) is still poorly understood however, and might require in-depth research and access to tacit knowledge relating to industry trends in order to predict. The time frame for returning to a software-based work and reappraising risk is therefore something which might need to be more regular for software than it has been for other forms of time-based media, at least as institutional expertise builds. In this section I reflect on the kinds of documentation generated during the active life of a work, particularly in relation to incidents of display and the application of conservation strategies.

3.3.2.1. Installation and Display

A new realisation of a work requires an understanding of the constellation of components which constitute that work, and of their respective significance. When a new realisation occurs, a reconsideration of the parameters of the installation is triggered, in light of technological change that has occurred since they were last formalised (e.g. at acquisition or for a previous realisation). This precipitates a revision of existing documentation to capture the structure of the new realisation and the nature of changes from previous realisations. The Guggenheim's "Iteration Report"

was developed by Joanna Phillips to meet this documentation need (Phillips, 2007). It focuses on describing a new realisation (Phillips uses the term “iteration”) of a work in terms of its components and their installation, while maintaining direct reference to the identity of the work through recording of deviations made from earlier realisations. The reasoning behind these decisions, and who made them, is also recorded, as well as reflections on the success of the realisation. While its principles remain valuable in the context of software-based art, the version of the report currently available from the Guggenheim operates primarily at the artwork installation level rather than the software level (Phillips, 2012). The elements that create the software performance are therefore not easily captured within this framework.

Research relating to the preservation of video games (McDonough, et al., 2010, Lowood, 2013) and networked artworks (Dekker, 2014, Guez, et al., 2017) has demonstrated the potential value (while also acknowledging the inherent difficulty) of maintaining a contextualised appreciation of cultural heritage as we move through time. Similarly, for software-based artworks there is a broader context to a particular realisation which it might be desirable to capture. This might not be easily addressed with approaches to documentation that rely on inflexible document templates, and there have been a number of experiments with alternative approaches. The ‘net.artdatabase’ project, for example, captures video footage of the experience of an individual interacting with an internet artwork (including the computer system used and the surrounding desk space) and juxtaposes it with a screen capture of their interaction (Sakrowski, & Dullaart, 2018). Narratives, or account-based descriptions of an artwork realisation, may supplement such documentary media. During a research residency at the Daniel Langlois Foundation in 2007, Lizzie Muller experimented with the use of oral histories to capture the experience of visitors’ interaction with an installed artwork (Muller, 2008). A narrative approach has also been explored by conservators at San Francisco Museum of Modern Art (SFMOMA). There, complex (and media rich) accounts of new realisations of time-based media artworks are generated in a collaboratively compiled document called a “technical narrative” (Hellar, 2013, p.3)—the inspiration for the title of this thesis—which is managed using a flexible Wiki system (Johnson, 2016).

In addition to describing the realisation itself, there are questions over documenting the relationship between the various realisations of an artwork, and between the digital objects (and environments) which constitute its software element. Approaches to describing such relationships have been explored through the repurposing of

models from bibliography—including examples from the media art conservation (DOCAM, n.d.) and video game preservation (McDonough, et al., 2010) domains. However, it remains to be seen whether these models would be fit-for-purpose in the context of describing software-based artworks, particularly given their layered nature—realised artwork on one level, and within that a software performance (ideas introduced in Chapter 2).

3.3.2.2. Preservation Strategies and Treatment

Whether associated with a display event or simply occurring within the rhythms of collection care, applying a conservation strategy or treatment is a major event in the course of a work's life. The basic intent of any conservation or preservation strategy is either to mitigate risks relating to future obsolescence or degradation, or to solve specific problems with the work as they arise. There are a number of general preservation strategies for time-based media art with applications to software-based art, all of which address these aims in slightly different ways. I will use Rinehart & Ippolito's classification from the monograph *Re-Collection*, which proposes: emulation (with which I include virtualisation), migration and reinterpretation (Rinehart, & Ippolito, 2014). Choosing an appropriate strategy is not a case of selecting a single pathway: they may be used together in a hybrid approach to preservation, which involves their combined application either in conjunction or at different stages in the course of long-term preservation. Reinterpretation, for example, is something which is necessitated to some degree whenever a work is realised through the necessary interpretation of installation parameters. Nonetheless, these three strategy types serve to illustrate the variety of ways in which preservation strategies are influenced by documentation availability and how they shape documentation requirements.

This first strategy I will cover is emulation, for which two uses can found in the literature. The first was championed by the Variable Media Initiative project and characterises emulation as the creation of "a facsimile of them [the digital and physical constituents of an artwork] in a totally different medium" (Depocas, et al., 2003, p.51). This essentially describes the simulation of an artwork through any suitable means—technical or non-technical. The second usage refers to a set of technologies which involve the use of software to mimic (hence *emulate*) a technical environment—typically hardware—in which the software can be executed. This theoretically allows for close approximation of original behaviour by recreating the precise requirements on hardware sometimes presented by software-based artworks and their execution

environments. Within this thesis I use emulation only in this context. Virtualisation, although similar to emulation in principle and result, involves a slightly different mechanism. Rather than imitating the target system's hardware completely, virtualisation allows an encapsulated software environment access to real hardware components (usually limited to the CPU). The limitation of this is that virtualisation software can only run (guest) environments which are supported by the native (host) environment. In allowing direct access to the processor however, virtualisation software allows the hosted environment to perform much closer to native speed (Rechert, et al., 2016).

Both emulated and virtualised guest environments can be considered semi-portable, in that they can encapsulate all dependencies and be run on any machine which can run the emulation or virtualisation software. This abstraction from underlying hardware reduces the impact of changes in the hardware environment, so lowering obsolescence risk. They also have the advantage of preserving something of the context of software by maintaining the look and feel of its software environment. Both techniques have found applications in the preservation of software-based artworks (Lurk, 2008, Falcão, et al., 2014, Rieger, et al., 2015, Rechert, et al., 2016). Recent work in the field of emulation presents significant new possibilities for providing access to emulated born-digital software-based art over the internet, using so called *Emulation as a Service* (EaaS) technologies (von Suchodoletz, et al., 2013, Rechert, et al., 2013). Despite their power, both emulation and virtualisation rely on specialised software which may bring with it its own set of preservation problems (although these are likely to be lessened), and in some cases legal considerations (Rosenthal, 2015). Applying either of them also requires detailed technical documentation about the native environment, for which there are currently no widely agreed upon templates or standards. In particular, it is crucial to understand the dependencies of the software in order to be able to reconstruct an appropriate environment to support it. It is also important that the parameters of a software performance are verifiable using suitable metrics or reference materials, yet approaches to documenting these parameters are also currently absent.

Migration, rather than attempting to maintain an appropriate execution environment, involves recreating the object of preservation using a contemporary technology. For software, this would involve re-writing the code for a contemporary platform. While a common preservation approach for digital materials (for example, digital video and research data), migration is uncommon in software preservation. While in some cases

this may be simply because it is unnecessary to carry out migration, it is also a resource intensive process and requires considerable care to be taken in the replication of the original artworks function and behaviour. In other cases, the actual code may not be available, so necessitating resource intensive reverse engineering processes. One such effort is recorded in Ben Fino-Radin's account of the restoration of Teiji Furuhashi's *Lovers* by conservators at MoMA (Fino-Radin, 2016). This complex work was painstakingly analysed and documented in order to provide the blueprint from which to rebuild the software at the centre of a control system and verify its performance in relation the other components of the work. In other cases, such as the Guggenheim's restoration of *Brandon* by Shu Lea Cheang (Phillips, et al., 2017), migration is partial: while some elements of the website remained operable with current web technology, others required updating. While both emulation and migration strategies require documentation from which to verify their performance, migration strategies require documentation of the functionality of the software, rather than the nature of its technical environment. Developing guidelines for capturing both of these early on in the life of an artwork is likely to provide the most value in terms of applying migration strategies at a later stage and avoiding the need to deal with difficult legacy issues in the future.

Reinterpretation is the final and perhaps most radical of the preservation strategies I will discuss—and there are very few case studies where it has been applied. This strategy relies not on maintaining the integrity of the original components of the artwork, or even its original functionality, but rather on a careful interpretation of the identity and intentions behind the work. This relies on a definition of the identity of an artwork that exists separately from its materials, a notion formalised by the Variable Media Initiative in the early 2000s (Depocas, et al., 2003). In practise, this means having the required artist support, rights, and documentation in place to enable the recreation of the artwork using new materials and techniques as necessary. Reinterpretation hinges on the idea of separating an artwork from the technology of its realisation, an idea which I introduced earlier in this thesis. Where this is possible within the parameters of the artist's intent or indeed, where it is carried out in collaboration with the artist, its application would require that parameters of change can be understood and justified within the lineage of that work.

One example of this strategy would be the iterations of Julia Scher's *Predictive Engineering* series of site-specific installations at San Francisco Museum of Modern Art (SFMOMA), each of which has involved a reinterpretation of the previous versions

of the work in order re-situate the original ideas of the artwork in a contemporary context (Clark, et al., 2015). The software at the heart of the installation was of course rewritten to support the new requirements that emerged during its development. All of this was carried out in very close collaboration with the artist, who has built reinterpretation into the work as part of its identity. While an unusual case such as this one suggests the potential for an expanded practice of conservation, in cases where the artist is not able or willing to engage in such work, strategies of reinterpretation may risk the loss the characteristics that constituted the artwork's identity. Where it is possible, there are questions over how the nature of these changes (and indeed, conservation treatments more generally) might be conveyed to museum audiences. Decrying the "cramped conventions" of the wall label and museum cataloguing systems, Jon Ippolito suggests that these approaches might fail to convey the "strange or complicated territory" that the realisation of a media artwork represents (Ippolito, 2008). How exactly such stories might be conveyed to museum audiences remains an open question however, and one which I return to later in this thesis (see Chapter 6).

In concluding this section, it is important to emphasise that there is no evidence that there will ever be a one-size-fits-all technical solution for the conservation of software-based art. There is great potential in bespoke approaches to conservation involving combined elements of emulation, migration (or modification) or reinterpretation: achieving the various realisations of *Predictive Engineering* has involved all three, for example. However, none of these strategies is straightforward to apply, and all rely on an in-depth technical understanding of the artwork's function and structure, and a nuanced appreciation of the artist's original intent and the parameters of the work's performance. Each of the strategies also places particular emphasis on certain aspects of documentation principles already introduced in this chapter—all of which remain poorly understood for software-based artworks. Emulation demands detailed documentation of technical environments; migration requires in-depth knowledge of the functionality and behaviour of the earlier version; while reinterpretation can only be carried out with a nuanced understanding of the artworks historical context.

More fundamentally, there are currently no published frameworks for how to record information regarding the application of a strategy or treatment to a software-based artwork—something core to the ethics of conservation. The AIC's Code of Ethics and Guidelines for Practice specify that:

> "During treatment, the conservation professional should maintain dated

> documentation that includes a record or description of techniques or procedures involved, materials used and their composition, the nature and extent of all alterations, and any additional information revealed or otherwise ascertained." (American Institute for Conservation of Historic and Artistic Works, 1994, p.9)

What a record or description of techniques of emulation, virtualisation, migration or reinterpretation might look like for software-based art however, has yet to be established within the conservation community. While the aim of this research is not to propose documentation templates for describing such treatments, the outcomes are likely to assist those developing them within their own organisations.

3.3.3. Information Systems

In the previous two sections I have discussed a number of activities within the conservation workflow and their implications for the collation and creation of software-based art documentation. In this section I take a slightly different perspective and examine a framework which sits in parallel to all phases of the workflow: the collection information system. Collections-related information systems within museums and archives are the means by which knowledge about a collection is managed, retrieved, manipulated and potentially shared. The information system is on one level a technical consideration, as it resides in the technology which enables information storage and access. The precise nature of the systems used on a technical level varies depending on the institutional context and history and may be found under various guises such as Collection Management Systems (CMS), Digital Asset Management Systems (DAM) and Digital Repositories (DR). While emerging from different cultures and with slightly different purposes, information systems are unified through their common use in capturing information about objects (be they digital or physical) which can then be manipulated in some way.

The forms of information that these systems might capture, typically highly *structured* information, is very much a documentation concern. The structure of this information is key to its utility, yielding possibilities such as search and retrieval, machine actioning, and the potential for sharing and exchange. Examining the collection information infrastructure at Tate, information systems are employed in the service of conservation activities in a variety of ways:

- Management of physical and digital objects, including tracking of their locations and recording of loans, and their relationship with an artwork and its realisations through time.

- Serving information to support analysis of and reporting on the characteristics of the collection or a subset of the collection.

- Allowing computer systems to manage and manipulate digital objects stored in a repository.

What these activities have in common, is that their value is contingent on the availability of structured information objects that to some degree *represent* the artworks, components and digital objects that they indexically relate to. I will refer to these as *structured representations*.

The dominant form of structured representation in museum information systems is *metadata*. With origins in libraries and archives, the term metadata developed during the formalisation of information science as a discipline, and can be understood "as 'structured data about an object that supports functions associated with the designated object'–with an object being 'any entity, form, or mode for which contextual data can be recorded'" (Greenberg, 2005, p.20). Although the terms usage is less frequent within the history of museum collection management, its principles are nonetheless ubiquitous within these environments. Here, metadata can be understood as structured data about a collection object. There are two possible meanings to the word within this context. The first relates to metadata *instances,* which are the concrete pieces of recorded information (such as an integer or a text string). The second relates to how these instances are structured and defined, and might be understood through a defined metadata *schema*. For example, 'Year of Creation' might be an element within a defined metadata schema for describing artworks, with a specific metadata instance for a particular artwork of '2008'.

Modelling and ontologies offer a logical extension to the principles of the metadata schema, allowing the structuring of a whole domain of knowledge and its formal semantics (Liu, & Özsu, 2009). These approaches may provide a means of developing sophisticated metadata representations in relation to particular domains of knowledge (Munir, & Sheraz Anjum, 2018). The uses of metadata for describing collections of time-based media artworks are well established (Fino-Radin, 2011, Rinehart, & Ippolito, 2014, Griesinger, 2016) and irrespective of the suitability of the models according to which the metadata is created, such artworks continue to enter institutional information systems. In practice, a metadata record for an artwork is a primary port of call when a work is revisited for purposes of exhibition, loan or study, and acts as a nexus for locating information on an artwork's constituent components

and their locations within storage.

Software-based artworks (and indeed, time-based media artworks in general) do not fit easily into these existing frameworks, due primarily to their structural complexity and multiplicity (see Section 2.6 for an explanation of these characteristics), both of which are difficult to represent using approaches to structured representation that are designed to address artworks as single objects (e.g. forms which are not realised or performed, such as painting or sculpture). A substantive structured representation (i.e. one which is useful and meets the purposes I outlined above) must be based in a clear conceptual model of the component types that constitute the software structures of a software-based artwork; including how they relate to each other, to the artwork as a whole, and to realisations of that artwork. While this would be valuable simply in supporting the software performance model developed within this research, there are also direct practical uses for such a model. During this research, issues relating to the representation of software-based artworks within collection management systems have been under discussion at Tate, while interviews with other practitioners reveal that similar issues have been faced at other institutions (B Fino-Radin, personal communication, 17 June 2016; J Phillips, personal communication, 12 December 2016; G Wijers, personal communication, 13 December 2016). Gaby Wijers, reflecting on the value of such approaches, points out that while “you can make an ideal metadata set […] then you also have to take in to consideration how much work needs to be done to fill it in” if it is to be pragmatically applied (G Wijers, personal communication, 13 December 2016). An appropriate system of structured representation will need to address this balance of completeness and usability.

There is no clear existing standard or model for creating structured representations of software-based artworks. If a suitable approach is to be identified, a number of existing approaches will require further exploration based on information derived from case study analysis. Based on a survey of published approaches, I have identified a selection with coverage that intersects with the concerns raised in this section. These are: Media Art Notation System (MANS) (Rinehart, 2007), PREMIS (PREMIS Editorial Committee, & others, 2015), Capturing Unstable Media Conceptual Model (CMCM) (V2_Institute for the Unstable Media, 2003), outputs of the EU FP7 PERICLES project (Waddington, et al., 2016) and CIDOC-CRM in conjunction with CRMdig ((Enge, & Lurk, 2014). Unified Modelling Language (UML) is another approach which, while not being intended for metadata specification, may also have relevance here given its

close relationship with software engineering and its suitability for describing software structures. I will critically appraise the potential use of each of these approaches in Section 4.6.

3.4. Documents for the Conservation of Software-based Art

Arriving at the end of this chapter, I have now developed the two halves of a conceptual framework (the first being developed in Chapter 2) which comprehensively describes the problem space this research seeks to address: how to effectively document software-based artworks in a conservation context. This concludes stage three of the constructive research methodology outlined in Chapter 1. From the analysis carried out in this chapter it is clear that, when considered in relation to software-based artworks, there are a number of gaps in existing conservation documentation practice. It is these gaps that I will address in the following chapters, which represent three distinct topics: analysis and representation of software structures; capturing the identity of a software-based artwork; and describing software evolution and version history. The rationale behind each of these is summarised below. Although the chapters are presented in an order by necessity of the document format, the research strands that resulted in these chapters were conducted in parallel. They are intended as both stand-alone solutions to the practical problem this research seeks to address, and as complementary components of a larger and more comprehensive framework. I demonstrate the applicability of each solution within each chapter using evidence from the study of case study artworks in each chapter (completing Stage 4 of the methodology), while research contributions and scope of applicability are addressed in Chapter 7 (completing Stage 5 and 6 of the methodology).

While research has resulted in a greater understanding of the documentation materials that might be sought when a software-based artwork is acquired, the significance of source code as the primary document raises questions over what actions might be taken if source code is not available. There is, therefore, a need to further develop approaches to examination at the software level, particularly those which can bypass the barriers to addressing compiled software. This is likely to be particularly significant in condition reporting processes, but also has strong synergies with the need to document individual realisations of software-based artworks. Furthermore, there is a need to consider the ways in which information derived from such analyses might be captured and incorporated into information systems, with reference to the array of competing metadata standards. In Chapter 4 I develop a

methodological framework for analysing software structures which complements source code based approaches, and explore the use of systems of structured representation in capturing this information.

Within all of the treatment strategies discussed—and sometimes between realisations of a work for exhibition—a degree of change in the original software super-object and its technical environment might be necessitated. This leads to questions over how to ensure that the identity of the artwork is maintained between realisations and versions. Through research over the past decade, parameters of acceptable change in time-based artworks and digital objects are now much better understood. However, this remains a complex area which the conservator must navigate individually for each artwork. Software performances present another layer to consider and one for which a formal framework has yet to emerge. In Chapter 5 I develop documentation strategies to assist in the capturing and managing of the identity of a software-based artwork at both the artwork and the software level, and so aiding decisions regarding its future realisation.

When change occurs in how a software-based artwork is realised, this change should be captured in documentation in order to fulfil the requirements of conservation best practice. On a structural level, there is a need to capture how the new version or realisation relates to the artwork as a conceptual entity. On a processual level, there is a need to capture or describe the changes made in a meaningful way. Finally, on a conceptual level there is a need to understand why choices were made and how they relate to the meaning of the materials employed. If the artwork is never truly fixed, these documentation materials present a crucial trail of evidence and historical insight into the life of the work. In Chapter 6 I explore how we might approach documenting change in the long-term care of software-based artworks, while ensuring the complex and evolving relationships between artwork, version, material and meaning are maintained.

CHAPTER 4

ANALYSIS AND REPRESENTATION OF SOFTWARE STRUCTURES

4.1. Chapter Outline

The purpose of this chapter is to examine approaches to the analysis and documentation of software structures, and to ascertain how they might most effectively support the requirements of the conservator. In Chapter 2 I introduced the idea that software-based art is typically structurally complex—that is, the arrangement of the parts of the artwork and the relationships between them can be many and varied. This applies not only at the artwork level, but also at the level of the software performance itself—the latter of which is currently poorly understood in a conservation context. I also highlighted the potential opacity of compiled software—the obfuscation of underlying code and process—which may make the comprehension of this structure particularly challenging. As proposed in Chapter 3, understanding and describing these software structures is an important component in planning the long-term preservation of the work, particularly in identifying how the software might be reliably realised in the future and in identifying components at risk of obsolescence.

In the first part of this chapter I introduce a simple workflow for the examination of

software-based artworks, which provides a framing for the discussion to come. In the next part I introduce ideas relating to software maintenance and reverse engineering in order to help situate software analysis within the frameworks of software engineering and clarify some of the more important concepts. I then explore in more detail the role of source code analysis as it has developed in conservation, including and its limitations, and then consider alternative and complementary approaches to the analysis of software structures. Taking established methods from software engineering, debugging and reverse engineering as a starting point, I assess their potential relevance and the limitations of their application, particularly in relation to the priorities of identifying the constituents of a software super-object and its relationship to its technical environment. In the last part of the chapter, I consider how the results of analysis might be formalised as structured metadata for incorporation into information systems. This takes the form of a conceptual model with mappings to several other relevant standards, designed to capture key information about a software-based artwork's realisation, its software components, and their relationships with the supporting technical environment.

4.2. Reconstructive Analysis of Software and Environment

As discussed in Chapter 3, the purpose of the examination and documentation of time-based media artwork realisations is reasonably well understood. Conservation workflows in this area of practice are carried out with the aim of gaining knowledge about an artwork's components and their meaning, the requirements for the works display, and how it might be cared for in the long term. Given that for software-based artworks another layer of realisation is present below that of the artwork—the software performance—we need to consider how to formulate examination and documentation processes at this level. In this section I will introduce a generic workflow for approaching the examination of software-based artworks, as a framing device for the analysis that follows. This workflow stems from research at Tate in 2016 in collaboration with Klaus Rechert at the University of Freiburg and Time-based Media Conservator Patricia Falcão. This project developed a tentative workflow for applying emulation strategies to software-based artworks (Rechert, et al., 2016)[13]. Many aspects of this workflow are applicable to a general analytical approach to deriving knowledge from software for purposes of examination and documentation—that is,

[13] The author of this thesis was a minor contributor to the project and an editor of the resulting report.

they might be used even where emulation is not applied. Taking this approach as a basis, I have formulated a less emulation-specific derivation of this workflow which I present here.

In essence, the workflow uses non-invasive disk imaging in combination with emulation and virtualisation technologies—principles which were introduced in Section 3.3—to reconstruct a technical environment (composed of an interlinked hardware environment and software environment) in which the software-based artworks can be executed. The disk image (a file-based encapsulation of data that might traditionally have been the contents of physical storage media) might be taken from a source computer system or manually constructed to create an appropriate software environment. The emulation or virtualisation tools provide the hardware environment. The primary purpose of the workflow is *reconstructive analysis*—the process of reconstructing a software performance as a means of identifying its parameters. If the reconstructed performance can be verified against the original, we can be more certain that we have identified the crucial components and their configuration. A secondary purpose is the production of a generalised (i.e. in which dependencies are made more abstract) and encapsulated representation of the software super-object and its environment which is portable and can be used for further study (e.g. studying the software's function and behaviour). Many of the issues touched on here are discussed extensively in Rechert et al., which also incorporates the rationale for the development of the workflow on which this research builds (Rechert, et al., 2016).

The workflow is presented in five stages below. The assumption is made that a digital representation of the software is available, and that preliminary information gathering (discussed in Section 3.3.1.1) has been carried out to some extent.

1. **Identify the software super-object that is being acquired.** This is the artist-approved version of the software which is intended for use in the display of the work. It could be acquired in a multitude of forms, such as installed on a pre-built computer system for display, stored on a USB flash drive, or delivered as a compressed bundle from an online server. In some cases, the software may be acquired with other supporting software dependencies.

2. **Use an appropriate non-invasive methodology to capture the software super-object (and software environment if applicable).** While in the most

straightforward cases this involves simply downloading files and verifying their contents, in cases where physical storage media are involved (or even provided as a functioning system of physical hardware components) this will necessitate the use of forensic disk imaging tools to ensure the integrity of the source data and the data captured.

3. **Examine and analyse captured software super-object, its environment and any gathered documentation (including source code if available) in order to identify technical environment components and configuration.** The primary objective of this stage is to gather as much information as possible towards rebuilding an appropriate technical environment for the performance of the software super-object. This stage may be iterated if are problems encountered in steps 4 and 5. Interaction with originals would be avoided where possible, but where necessary careful consideration should be given to the risk of interacting with them.

4. **Reconstruct technical environment using captured software super-object, gathered information and any required software or hardware components.** A physical computer system might be built or, more pragmatically in many cases, a virtualised or emulated hardware environment. If the software super-object cannot be run in the reconstructed environment, stage 3 is returned to in order to acquire more information and address the problem.

    4.1. **If possible, reconstruct production environment and attempt recompilation of software.** This extra step provides additional assurance that where software production materials have been acquired, they are complete. Furthermore, if it is ever necessary, modifications could be made and the software recompiled.

5. **Verify the reconstructed software-based artwork performance against an artist approved version or suitable documentation, in collaboration with the artist or other authorised person where possible.** The aim of this stage is to ensure an authentic realisation of the artwork and might involve side-by-side comparison with another version, testing and measurement, and the conservator’s own judgement. Where the performance is found to be inadequately representing the original, stage 3 is

returned to in order to gather more information.

5.1. **If possible, create a generalised, portable version of the technical environment using emulation, virtualisation or containerisation technology.** Selection of the technology to be used will depend on the available tools for meeting the technical environment requirements of the software. The encapsulated version generated can act as both a valid representation of the software-based artwork's software component and documentation of the reconstructed environment.

6. **Document the technical environment and configuration that achieved the verified performance.** The documentation of the composition of the software structure that resulted in a successful performance provides important documentation for achieving future performances, and thus realisations of the artwork itself. In practice, much of this documentation work may occur alongside the previous stages.

The reconstruction and verification of software performances in this way, would—through the isolation of an appropriate technical environment—develop considerable insight into their technical basis. This process also presents other advantages. The accumulation of reusable software components (e.g. runtime libraries or drivers), pre-built disk images (e.g. a generic installation for a particular operating system) and tools (e.g. analysis tools or virtualisation configurations) means that undertaking the process for other software-based artworks in the future may be simplified. There is also the potential for elements of the workflow to be automated, for instance where similar kinds of software are encountered. A workflow reconstruction tool such as Apache Taverna (anon. Apache Taverna, 2016), for example, could be used to (partially) automate an analysis tool chain.

The process of reconstructive analysis outlined in this section offers a framework for understanding how the examination and documentation of software-based artworks might be undertaken. However, there is currently a lack of research in two key areas, on which its usability is dependent: methods for the analysis of software and environment; and approaches to recording the information gathered in a way in which it is useful for conservators. In this chapter I aim to address these two gaps. Analysis primarily occurs during stage 3 of the workflow with the aim of information gathering but may also occur where a performance is verified at stage 5. I explore approaches

to software analysis in Section 4.3 to 4.5. Stage 6 of the workflow implies a need for some system of representation with which to capture the insights gained from analysis regarding the software super-object and its relationship with its technical environment. These are likely to be particularly significant in relation to the demands of institutional information systems introduced in Section 3.3.3. I explore and develop methods for deriving such representations in Section 4.6.

4.3. Legacy Systems and Reverse Engineering

Software analysis and documentation are not new fields—indeed, while these processes have only become of interest to conservators relatively recently, their history parallels that of software. Therefore, contextualising these processes within the field of software engineering is a helpful starting point to this discussion. In many cases, software-based artworks fit within the software engineering conception of a *legacy system*. Legacy systems can be defined as socio-technical software systems (that is, they involve both technology and existing users or business processes) which rely on languages or technology components which are no longer current (Sommerville, 2015, Butterfield, & Ngondi, 2016). Alderson and Shah acknowledge that that there is little real consensus about when a system can or should be labelled 'legacy', and that this is usually a strategic consideration relating to the costs and benefits of maintenance (Alderson, & Shah, 1999). It is therefore important to understand something of what software maintenance means as a part of the software engineering lifecycle, and how it might relate to the goals of conservation.

Software maintenance relates to the totality of activities required to support an operational software system, particularly the modification of the system after delivery to correct faults, adapt to changes in environment and to prevent future operational problems (anon. ISO/IEC 14764:2006(E) IEEE Std 14764-2006: Software Engineering — Software Life Cycle Processes — Maintenance, 2006). While the typical focus of software maintenance is continuous delivery of system services, the conservator has analogous goals in relation to software-based art. Indeed, in their work on source code analysis in conservation, Deena Engel and Glenn Wharton highlight the significance of software maintenance in relation to the long-term care of software-based artworks (Engel, & Wharton, 2014). Legacy systems pose a particular challenge to software maintenance in cases where some kind of custody change has occurred, as is often the case for artworks acquired by an institution. In these cases, the conservator is in a similar position to the role of a new maintainer of a legacy system. Their primary goal is understanding the software: what it does, how it does it

and how it can be kept running.

To achieve this, one might look at the non-software product outputs of the development process—often referred to as *artefacts* in software engineering (although there is no widely agreed definition of this term). The potential nature of these materials was explored in Section 3.3.1.1, where I found that while it is possible to derive best practice guidelines from software engineering practice, extensive documentation in accordance with these guidelines is unlikely to be received when an artwork is acquired—certainly, this is case for the case study artworks examined during this research. Evidence from other research indicates that, while artists working with new media value documentation highly in relation to the legacy of their work, the documentation they generate is idiosyncratic, linked closely to their mode of practice and often concerned with shorter time spans than museums might be required to consider (Post, 2017). As such, the presence of this kind of detailed design documentation is hardly guaranteed, nor will it necessarily be suitable for the purposes of software maintenance and preservation activities. There are also risks associated with prior generated documentation presenting an idealised view of the artwork or one which differs from the final realisation of the work as it was acquired. The danger of documentation being out of date or mismatching the deployed software is an acknowledged concern regarding its value in a software engineering context (Lethbridge, et al., 2003). Therefore, conservators must be at the very least prepared to verify such documentation to some extent.

In software engineering, the challenges raised by a poorly documented legacy system might be addressed using *reverse engineering*. Chikofsky and Cross (1990) define reverse engineering as “the process of analysing a subject system to identify the system’s components and their interrelationships and create representations of the system in another form or at a higher level of abstraction” (p.15), and that it can can be considered in contrast to “forward engineering”, the traditional process of moving from design to a physical implementation. A term first formalised in 1985 by M. G. Rekoff in the context of “cloning” or recreating existing hardware systems (Rekoff, 1985), it has since come to encompass a much broader practice that includes deriving documentation that supports program understanding from existing software representations. Engel and Wharton demonstrate the value of a reverse engineering approach based on the analysis of the source code representation (Engel, & Wharton, 2014), the purpose of which (in relation forward engineering) is illustrated in Figure 6 below.

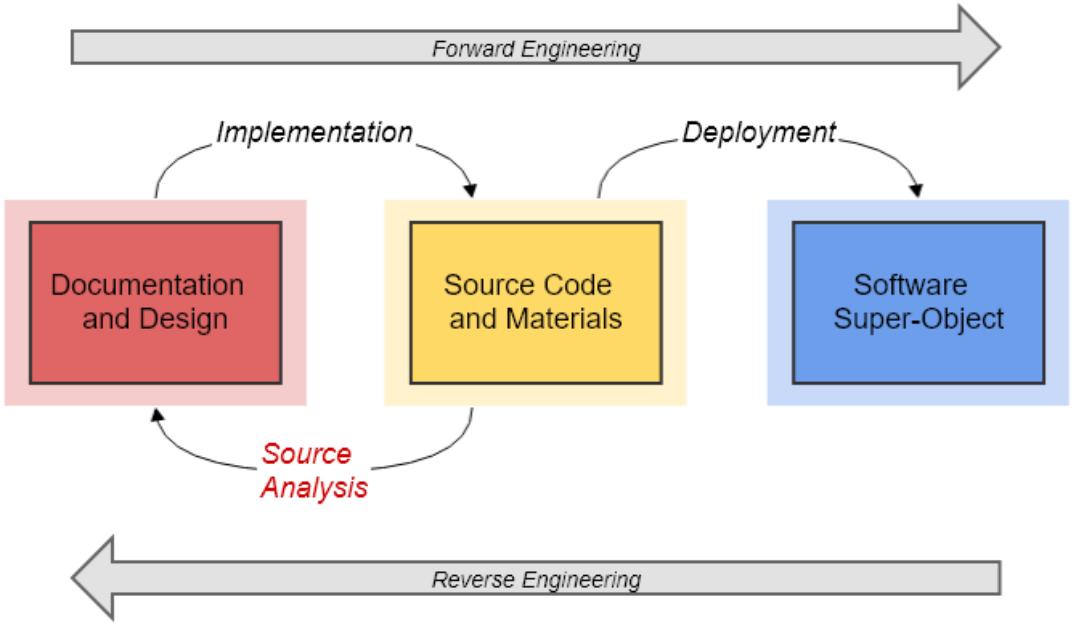


Figure 6. Representation of the forward and reverse engineering processes in relation to artefacts resulting from processes in software engineering. Arrows between boxes relate to processes of forward engineering[14] above (from left to right) and source code analysis as a method of reverse engineering below (from right to left).

Source code's value is also well established in the broader software engineering field, and research has found that software engineers consistently rate source code as the most important artefact produced by the software development process in terms of maintenance value (Singer, 1998, de Souza, et al., 2006, Das, et al., 2007). In Chapter 2 I identified a number of concerns regarding the limitations of source code analysis as an approach to documenting software-based artworks. These included scenarios where source code is unavailable for software or where it may be impractical or unnecessary to undertake such work. In the next section I examine these limitations in more detail.

4.4. Problematising Source Code Analysis

While I have largely made reference only to source *code* in the preceding section, it is more appropriate to consider *source materials*—a more general term inclusive of

[14] Implementation and deployment are used in a variety of ways in the software engineering literature. Implementation is used here to describe the process of moving from concept to code, while deployment refers to the process of making an implemented concept useable in its operational environment.

non-source code elements involved in the development process such as data and production software. As I have already illustrated, the value attached to source code within this body of development materials is high. A number of explorations of the analysis of source code by Deena Engel and collaborators have already revealed how this process can result in deep insight into the workings of software-based artworks and the creation of rich technical documentation (Engel, & Wharton, 2014, Engel, & Wharton, 2015, Dover, 2016). These source materials also present other benefits beyond software analysis. They are a trace of the process of artistic development and creation, and as such are significant artefacts in their own right and worthy of preservation as historical documents (ideas explored further in Chapter 6). Furthermore, if the complete environment containing the original set of code, data and tools can be reconstructed, it may even be possible to modify and recompile software, if desirable. While the value of having access to source materials is impossible to dispute, I propose that there are three factors which may mitigate the benefits of taking a source-centric approach to software analysis: *inaccessibility*, *nonequivalence* and *redundancy*.

Inaccessibility arises where source materials are not available for examination, or what is available is in some way an incomplete representation of the software. Perhaps most obviously, this problem might arise where source code is unavailable altogether, either because it never existed (for example, where WYSIWYG[15] development software was used) or because it could not be acquired from the artist (perhaps because it was lost or they simply do not wish to share it). The use of a complex development environment may also impact accessibility. For cases where source materials are simply plain text source code, accessibility is unlikely to pose a problem as the code can be easily rendered and preserved. For four of the six software-based artwork case studies addressed in this thesis, source materials consist primarily of plain text source code. However, while this source code underpins the creation of the software employed, the source materials in each case consist of more than just plain text source code. Integrated development environment (IDE) software and other authoring tools were used in each case to simplify elements of project management, programming (such as working with libraries and debugging

[15] This is an acronym of “what you see is what you get”, and is used here to refer to development software which uses visual interfaces to make the process of development more intuitive.

code) and interface design.

Without access to this complete development environment, some elements of the software in question may remain unclear. Even for source code-based projects where this is less essential, it can be a significant comprehension aid in providing structure to the various elements of the program, particularly where it is complex. If there is a desire or need to achieve recompilation however, ensuring access to a complete development environment is essential, preferably including the original versions of the software that made up that environment. In practice, recompilation may be an ambitious goal in many cases, and indeed, not necessary for preservation purposes (applying an emulation strategy for example, does not require recompilation). For all the artwork case studies, loading the source code projects into contemporary IDEs (where this was possible at all) resulted in errors which would need to be addressed before they could be recompiled.

Problems with accessibility may also manifest for software developed in IDEs and other production tools which create further abstraction layers between user and code. Many development tools abstract underlying complex systems, such as graph-based visual editors (e.g. Max for audio processing), WYSIWYG editors (e.g. Visual Basic for building forms) and 3D engines (e.g. Unity for developing video games). The Quest3D software used in the development of *Sow Farm* presents a clear example of this problem. This now obsolete software—it is no longer sold or supported by its developer Act-3D—simplifies otherwise complex programming tasks relating to the 3D rendering pipeline through the use of a graph editor. Custom code can be developed within this environment, but this code alone would not be sufficient to understand the software super-object, let alone recompile it. Even *with* the complete development environment, reliance on obsolete, closed source technology adds significant additional preservation requirements if long-term access is to be maintained to these. Later in this chapter I consider binary-centric analysis approaches which can to some extent address problems with the accessibility of source materials.

Nonequivalence refers to the potential for the binaries included with a software-based artwork acquisition to have an unclear provenance in relation to the corresponding source materials in the same acquisition. This may arise for a number of reasons. In some cases, the source materials acquired may simply not be a complete representation of the materials involved in the creation of a particular set of binaries, for reasons of accidental omission or loss. In such cases it is therefore not

possible to make fully informed inferences about the binaries on the basis of the source materials. This might also be a problem where binaries were generated within the original development environment, using a particular configuration that is no longer known. Without detailed documentation of the build process, equivalence can only be inferred by acquiring the complete development environment, recompiling the software and comparing performances in a suitable technical environment. This is a task which, as discussed earlier in this section, may not be possible if the examination is occurring a long time after the artwork was created, particularly taking into consideration the loss of associated tacit knowledge. They may also arise where alterations to software are being made rapidly (perhaps in relation to a deadline), resulting in a proliferation of versions that may have been inadequately tracked.

Problems with nonequivalence between binaries and source materials are evident in the examination of the *Brutalismo* software. This artwork has a large Java source code project associated with it, which was developed in the NetBeans IDE. Within the source code project, there are several sub-projects and a number of modules (function-related organisational structures for blocks of code) which were not incorporated into the binaries used when the software was built (i.e. transformed into an executable set of Java files). Furthermore, there are numerous versions of the binaries without a clear naming protocol, making it difficult to establish concrete links with the code base. These kinds of problem might be mitigated by communication with the artist on acquisition of a work, and where possible the associated acquisition of a development environment allowing recompilation of the software (although this may have significant licencing cost implications). Where custody of the code is still shared by artist and institution, change management and versioning can provide a means of ensuring that equivalence is recorded—ideas which are explored further in Chapter 6. Where this kind of collaboration is not possible, methods for reversing the compilation (or build) process may prove useful in establishing equivalence. In other cases, dealing with questions of equivalence may be avoided by analysing the binaries or process (at runtime) directly. I discuss these approaches in more detail later in this chapter.

Redundancy refers to the potential for source code analysis to be surplus to requirements during the examination of software by a conservator. This may arise where the effort required to develop program comprehension through code analysis outweighs the value of the information that might be gained, or where information might be more easily gained using an alternative approach. With increasing

complexity of source materials comes greater challenges to program comprehension. Program comprehension is aided by documentation: for example, a structural overview or documentation of the relevant programming language syntax. As highlighted in Chapter 3, the process of source code analysis is also eased considerably if the source code is well documented by in-line annotations (or code *comments*). Where these are missing, and resources are limited, and presuming the conservator is unable to carry out the analysis themselves, questions arise regarding whether to seek external expert assistance. While doing so is not unusual within conservation practice, in some cases detailed analysis of source code may simply not be necessary.

Given that the demands of the reconstructive analysis workflow introduced at the beginning of this chapter are quite specific—identifying the technical environment required to perform a software super-object—analysing source code may not be the most effective way of addressing them. This can be understood as another example of a map-territory problem (discussed in the context of representation in Chapter 3), in that a decision must be made about the value of an exhaustive approach versus a pragmatic one. When Microsoft released the file format specifications for their native Office XML formats, the specifications were found to be extremely large and complex compared to those for similar formats, so frustrating those interested in developing software that could use them and stymying interoperability (Hiser, 2007). Former Microsoft programmer Joel Spolsky, suggests that this relates to the complex history of the software they were designed for:

> “The bottom line is that there are thousands of developer years of work that went into the current versions of Word and Excel, and if you really want to clone those applications completely, you’re going to have to do thousands of years of work.” (Spolsky, 2008)

While it is impossible to know the actual amount of work that would be required to clone Word or Excel, the point of relevance here is that reverse engineering complex software is an inherently a resource intensive activity. Completely understanding a complex software-based artwork through its source code may take a considerable amount of time. Therefore, the question we might ask prior to examining an artwork, is whether it is necessary or efficient to carry out source code analysis as part of this process. In cases where an environment-centric preservation strategy is applied (such as emulation or virtualisation), understanding the intricacies of the software programs functionality is not helpful.

Practical examples of this problem are, again, easy to find among the artwork case studies. Rafael Lozano-Hemmer's *Subtitled Public* uses very complex software consisting of over 60000 lines of code written in an old version of the Delphi programming language (a derivative of Object Pascal). If the conservator wants to understand how to prepare a new technical environment for display, analysing the binaries directly is much more efficient than consulting the large volume of code, as this permits targeted extraction of such information without any requirement on the conservator to be able to read the programming language used.

Indeed, it may also be less ambiguous, as in other cases redundancy may stem from the fact that source code does not accurately capture connections with technical environment. Looking at the source code of *Colors* for example, it is impossible to concretely identify dependency relationships with the QuickTime framework through the source code alone. We can find calls to QuickTime libraries, but we don't know whether these calls would work for all or only a subset of the released versions of QuickTime. In this case, understanding the software involves also understanding the complex development history of QuickTime, a closed-source, proprietary framework maintained by Apple. Approaching the problem of dependency management pragmatically, we might instead test the application in different software environments with different QuickTime versions, and so establish the parameters of its portability. This reflects the fact that each computational process (which exists in memory only during the period in which it is executed) is unique and ephemeral, and not equivalent to the binary or the source code. This brings us back to issues of nonequivalence: the software process through which a software performance is generated is not equivalent to the source code representation of the software being executed in memory.

It is important to note that certain models of artist-institution collaboration in the care of software-based artworks dramatically reduce the risks posed by the factors discussed above. This includes relationships that are either closely collaborative or involve sharing infrastructure prior to or after acquisition. Collaboration with artists and programmers has been a common approach in the care of software-based artworks at Tate. During the installation of Rafael Lozano-Hemmer's *Subtitled Public* at Tate Liverpool in 2008, the software was altered during the installation process and recompiled. Similarly, work on the Jose Carlos Martinat's *Brutalismo* software was able to continue between installations, as the programmer remotely connected to a Tate hosted development machine. In these cases, it becomes feasible to generate

some of the essential code documentation in collaboration with the artist and other collaborators, where resources permit this—an idea I return to in Chapter 6.

4.4.1. Case Study: Program Comprehension Through Source Code Analysis

In this section I will examine the value of insights gained from source code analysis of the 2010 Flash version of *Becoming* by Michael Craig-Martin. The original version was developed in 2003 using Macromedia Director 8, an authoring tool for creating Shockwave multimedia applications, by London-based digital design company AVCO. With an interest in exploring how the process of migrating software to another technology might work, in 2010 Tate worked with the artist and AVCO to develop software using a similar contemporary software platform—Adobe Flash Professional CS5.5. The Flash version replicates the behaviour and formal characteristics of the original using a reimplementation of the code in the ActionScript 3 scripting language and a third-party extension library called GreenSock. While this version of the software has not yet been used in a realisation of the work, I chose to examine this version simply because it still runs correctly in contemporary operating systems (unlike the original Director version) and will therefore offer greater potential in terms of long-term preservation. Flash is technically on the cusp of obsolescence, with the technologies maintainer Adobe announcing that support will end by 2020 (Adobe, 2017). However, Flash projector executables (which are not dependent on a web browser) compiled for Windows operating systems still run natively on its most recent edition (Windows 10), without the need for additional supporting software. Loss of access is therefore not an immediate risk, although must remain under review in respect to the continued evolution of the Windows platform and PC hardware.

The work consists of custom software used to generate dynamic 2D graphics displayed on an LCD screen. This screen is housed in a custom-built case which provides framing and conceals the computer hardware. The 2D graphics are an assemblage of everyday objects drawn in Craig-Martin's signature style of brightly coloured line drawings, rendered against a magenta background. Elizabeth Manchester, writing for Tate, describes the dynamic elements of the software as follows:

> "For this project, AVCO developed a programme that generates the random appearance and disappearance of the objects. [...] The objects may all appear at once, or none may be visible for a considerable length of time. The programme allows for unpredictable combinations which may never be repeated." (Manchester,

2004)

The parameters of the algorithms which result in these “unpredictable combinations” are unclear from examining the artwork when it is displayed. The speed at which objects fade in and out could be measured manually using a timer, but these times are found to be variable, and the reasons for this (as well as for the selection of an object to add or remove) is impossible to determine from looking at the screen output alone. Certain behaviours do hint at features of the underlying algorithm. For example, when the software starts, all objects are visible, and the program initially seems to remove objects (randomly) at a much greater rate than it replaces them. However, more precise information that this would be extremely difficult to determine.

The ActionScript code used in this version of the work can be expressed as plain text, but in order to create the software it was combined with the graphical elements of the work within the Flash authoring software. The source code consists of four ActionScript files with different purposes:

- Becoming.as: Initialises the animation and instantiates the BecomingView and BecomingController classes;
- BecomingView.as: Instantiates the BecomingObjects as layers;
- BecomingController.as: Controls the appearance and disappearance of objects within the scene;
- BecomingObject.as: Initialises the object fading animations and assigns a 2D graphic to each object.

The BecomingController functions are by far the most complicated part of the code and use nested conditional statements and pseudorandom number generation to create variance in the behaviour of the software. Studying the BecomingController code reveals a number of features of the underlying processes:

- While pseudorandom number generators are used in the creation of some variables (such as fade time and choosing which object to remove) they are all constrained in some way in relation to the current status of the animation.
- If there are more than 8 objects in the process of changing (from visible or hidden), no further changes will occur until this number reaches 8 or less.

- There is random wait time of between 0.4 and 3 seconds before checking whether to remove or replace another object.

- The most complex part of the code deals with deciding whether to remove or replace an object at the current time. This takes into consideration the number of objects currently visible, the number of objects removed, and whether the last action was a replacement or removal.

- The speed of a removal or replacement animation is linked to the number of currently changing objects. Based on the number of changing objects, a random time is chosen within a range that is specific to that number (these range from the lowest of 27.5 seconds and highest of 46.4 seconds).

This information greatly helps us understand the parameters of the *Becoming* software's functionality, and detailed documentation of the exact parameters could allow them to be recreated using another platform. However, it does little to tell us about the requirements of the software in terms of its execution environment. For example, which components of the host technical environment are utilised by the *Becoming* process at runtime and what impact they might have on the software performance. This problem stems from that fact that much of the Flash technology at the core of the work is not directly accessible to the user of the Flash development software. This is because these proprietary elements are simply not present in the source materials as far as the developer is concerned; rather, the Flash development software incorporates them into the binaries when they are generated for use. Developers are drawn to platforms such as Flash precisely because the out-of-the-box functionality they offer does not require them to build their own equivalent software from scratch, but it also means that the developers of the platform (such as Adobe, the current owners and maintainers of Flash) will typically keep their proprietary source code private.

As already stated, Adobe has announced plans to end support for and distribution of Flash by 2020 (Adobe, 2017). Therefore, from a conservator's perspective it may be sensible to think about migrating the *Becoming* software to a new technology before this time. In this case the artist has not indicated that the Flash rendering engine is conceptually significant to the work's realisation, so the most significant consideration would be how to maintain the software performance as accurately as possible. HTML5 and JavaScript technology offer an open and community-led standard that may offer a suitable migration pathway. Approaching this migration would necessitate

considering not only whether the ActionScript functionality of the source code could be reimplemented in JavaScript, but whether there are features of the Flash rendering engine which might need to be replicated. For example, the anti-aliasing of the edges of the vector graphics is handled by the Flash renderer and results in a particular quality of smoothing to their edges.

4.5. Binary-centric Software Analysis

An alternative to source code analysis is to instead look to the binaries; the compiled software representation. As discussed in Chapter 2, the problem presented by binaries in developing program understanding is that they are a relatively opaque software representation—their internal structure is complex and designed for machine comprehension, rather than human. Fortunately, there are other reverse engineering approaches which can be used to address precisely this problem. These are illustrated in relation to forward and reverse engineering, including source code analysis, in Figure 7 below.

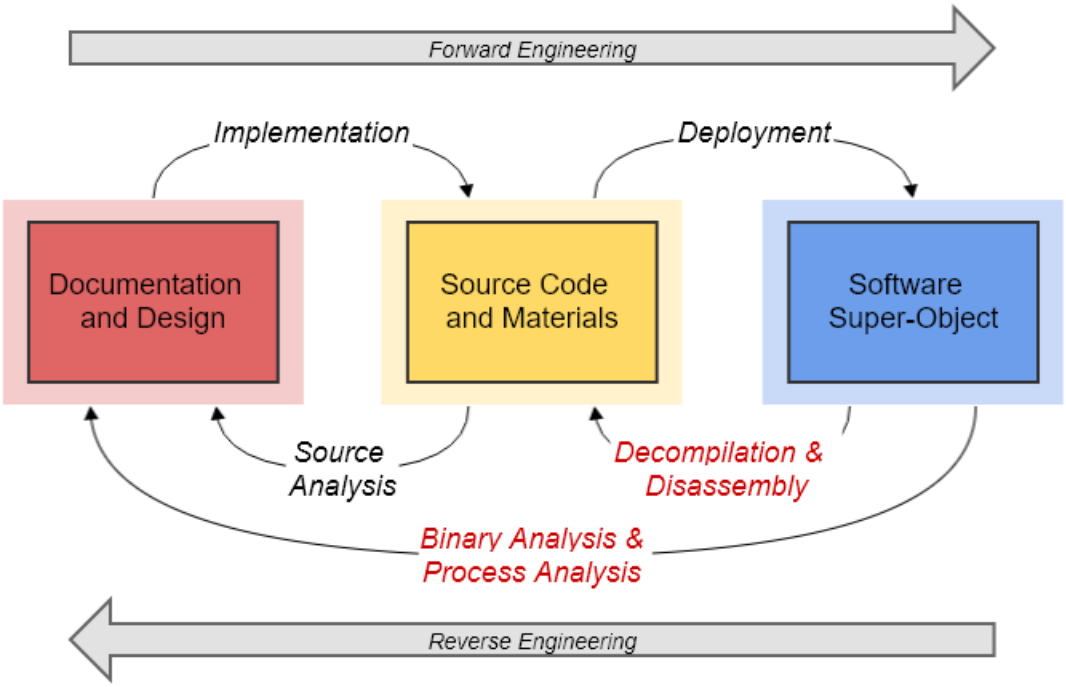


Figure 7. Representation of the forward and reverse engineering processes in relation to artefacts resulting from processes in software engineering, extended to incorporate binary-centric analysis methods. Arrows between boxes relate to processes of forward engineering above (from left to right) and reverse engineering below (from right to left).

Binary analysis seeks to analyse the compiled code contained within the binary files and may be potentially useful as a way of extracting information. Decompilation and

disassembly can transform compiled software back into a higher-level representation (i.e. something more analogous to the source materials). Given that the software performance is the result of a distinct computational process rather than of the software super-object as a static digital object, there is a need to address this process component too. The considered use of dynamic analysis tools can permit “the analysis of data gathered from a running program” in order to gain program understanding (Cornelissen, et al., 2009, p.684). I look at all of these techniques in more detail in the following sections.

4.5.1. Binary Analysis and Decompilation

Where source materials are not available, it may be possible to use reverse engineering methods to derive equivalent information from the binaries. One such approach is simply to analyse the content of the binaries. While the code contained in binaries is typically low level and intended for execution (or interpretation) by a machine, tools have been developed which can extract information from this code and from the other metadata stored inside such files. Tools for doing this are sometimes called *static* binary analysis tools and have found particular use in the identification of malware (Bergeron, et al., 1999, Moser, et al., 2007). In the context of examining software-based artworks for which source code is not available, binary analysis can extract useful information about the structure of the binary file. For example, it is a simple way to ascertain whether an executable was built for x86 or x64 processor architectures—a useful piece of preliminary information in identifying technical environment requirements. It also offers a powerful tool for identifying dependencies. In the example illustrated in Figure 8 below, the CFF Explorer (Pistelli, 2012) binary analysis tool has been applied to the calibrate.exe Windows Portable Executable used in the setup of *Subtitled Public* by Rafael Lozano-Hemmer. This enabled the capture of information about the software that was not available for the software when it was acquired, including the nested set of Windows Dynamic Link Libraries (DLLs) required by the program and metadata describing them. In the example pictured, the cv.dll library is a specific version of an Intel computer vision library.

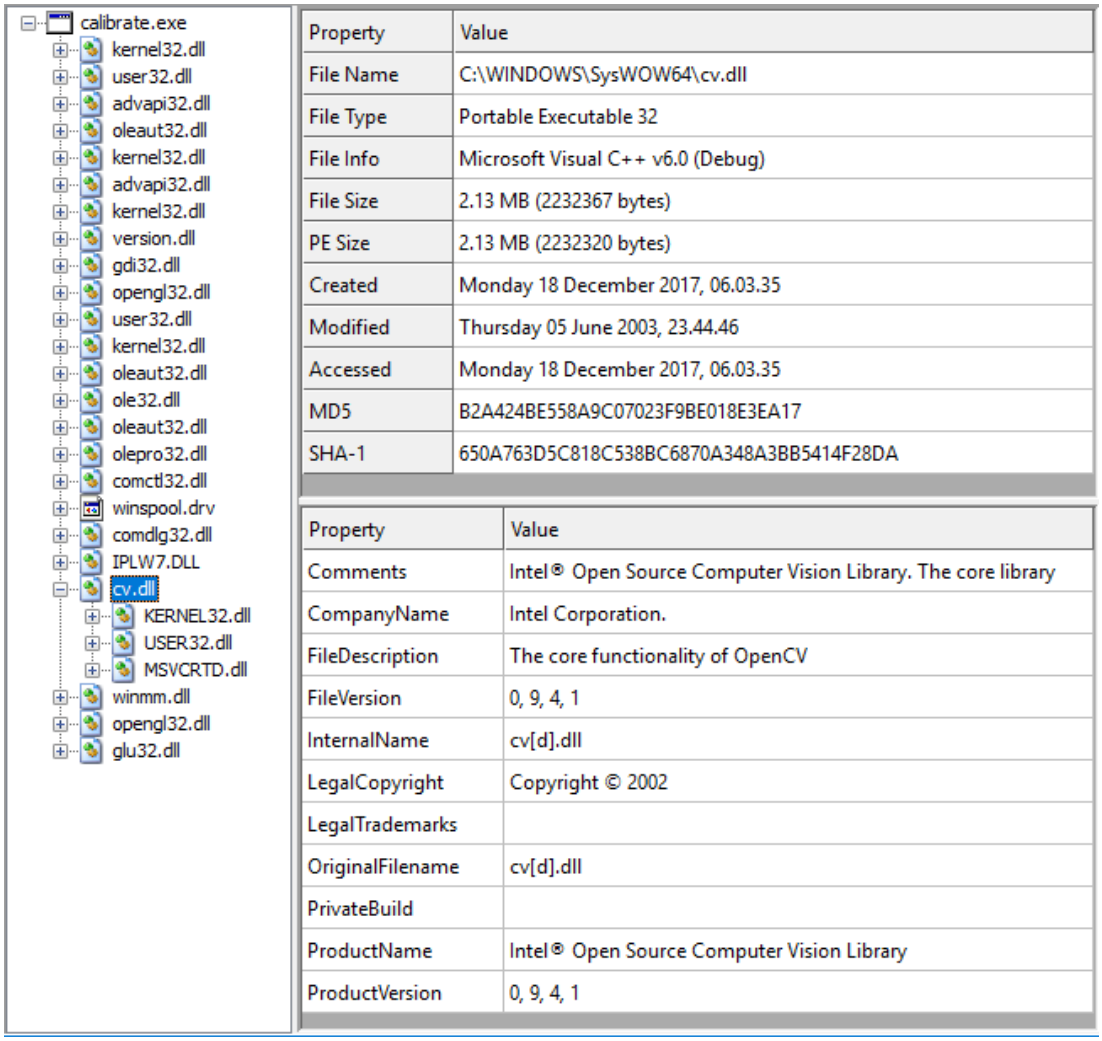


Figure 8. The nested DLL dependencies (along with metadata describing one of them) of the *Subtitled Public* calibrate.exe program, revealed through the use of CFF Explorer binary analysis tool. The third-party Intel OpenCV library is highlighted.

Another binary-centric approach is to attempt the transformation of the low-level code contained in the binaries into a representation at a higher-level of abstraction, which might be more easily comprehended by a human reader. The transformation from source materials to compiled software is essentially a one-way process, and so the original source materials can never be derived exactly as they were. However, as a representation of the program is still essentially present in the binary code (the exact level of abstraction varies depending on the language used), it may be possible to generate a higher level representation from it if a suitable tool is available. This process of reversing compilation is known as *decompilation* (Geffner, 2014). In relation to available tools, the term decompiler has a slightly more ambiguous

meaning, and might also refer to tools such asset extractors[16] which do not actually translate machine code. These are nonetheless relevant tools in seeking to derive source materials from binaries, as they might allow the extraction of data assets which are packed into an executable, resource file or other encoded form.

The nature of the transformations that occur during compilation mean that decompilation is not a straightforward process, and the efficacy and usefulness of decompilation tools varies considerably depending on the type of software being targeted. This can be demonstrated through its efficacy in relation to three of the software-based artwork case studies, each of which relies on a different software platform. [REDACTED] has associated source code acquired by Tate, which has been commented by the developer [REDACTED]. This allows for meaningful comparison with the results of decompilation. As [REDACTED] the program was written in Flash ActionScript, which is translated by the Flash runtime on execution, there is no need for the interpretation of machine code and we might therefore expect a high-level level of correlation between decompiled code and source code.

The source code was decompiled using a Flash decompilation tool called JPEXS (JPEXS, 2016). This program outputs a set of resources which correspond roughly to the assets that make up a Flash project, including the graphical data assets, animation data and the ActionScript code itself. In Figure 9 below, I compare compiled and decompiled versions of the same segment of source code from a class called [REDACTED].

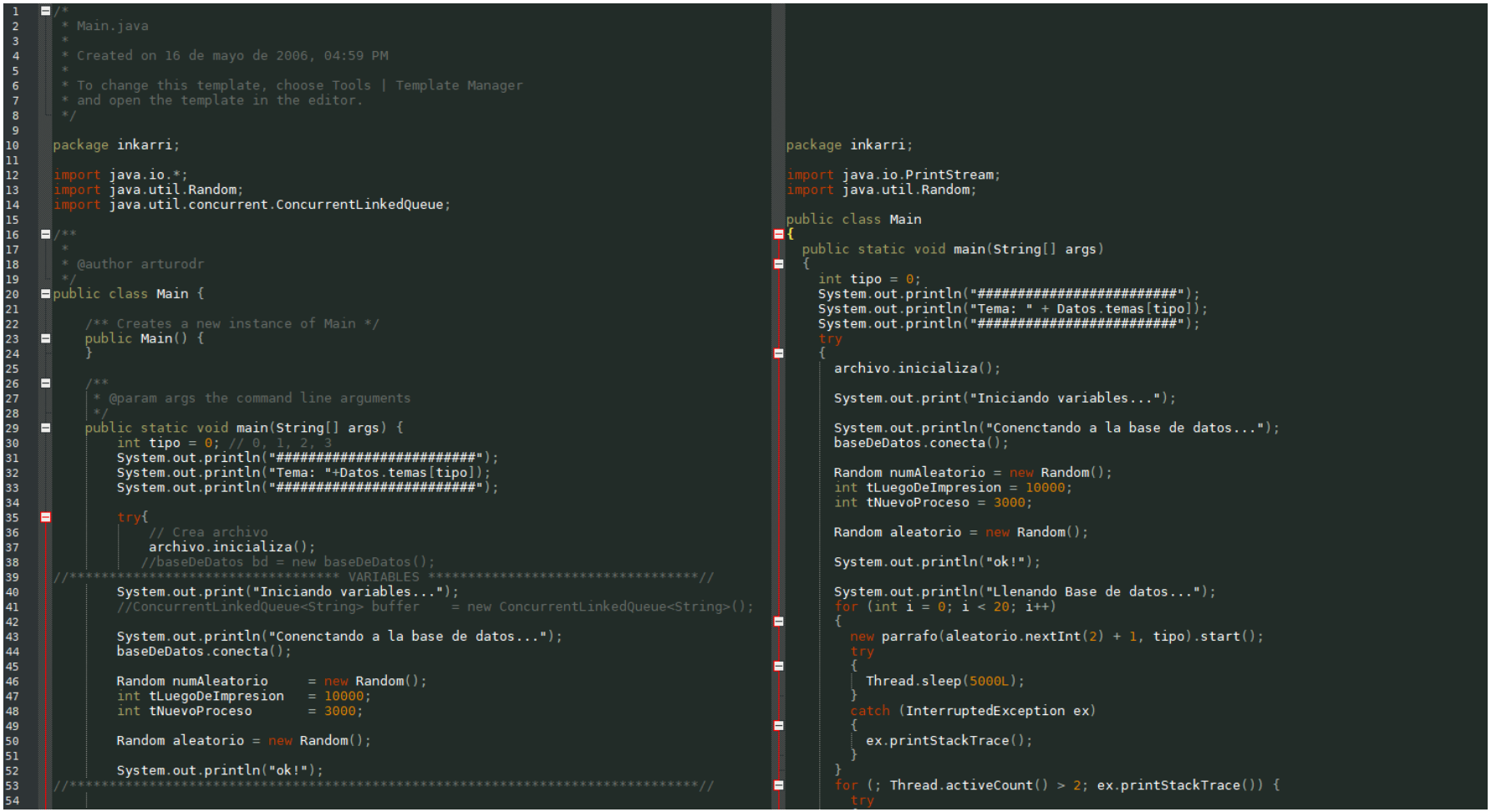
[16] Such tools might, for example, decode compressed packages of data (which are manipulated by a software program when it is executed) and so allow the examination of their contents.

[FIGURE REDCATED]

Figure 9. Comparison of a snippet of original ActionScript 3.0 source code (left) and decompiled code (right) for [REDCATED]. The decompiled code has been modified to include spaces where the header would be, to allow easier line-for-line comparison with original source code.

The content of the decompiled code is very similar to that of the original source, including consistent file, class and variable names. There are however a small number of changes apparent, including one package import being condensed into a generalised form and one variable name change. These would have a minimal effect on program comprehension. The missing comments (grey formatted text) from the original version however, are slightly more significant, most strikingly apparent in the absence of the metadata header describing the file, its author and version information. A common feature of compilers (and other build tools) is that they strip out code comments and metadata in this way. More subtly, while variable names have largely been maintained, line break formatting has not been retained, resulting in the loss of logical groupings of related variables. In this case, the decompilation output is clearly a useful representation of the program and could form an effective basis for the recoding of the software. However, the loss of comments and other authorial traces in the decompiled code means that program comprehension is slightly more difficult, and the decompiled code offers a rather less rich history of the development process.

The *Brutalism* Java binaries are another case where decompilation is likely to be successful, as Java is not compiled as machine code. Java binaries contain a representation of the code known as bytecode, a higher-level abstraction than machine code, and one which requires interpretation by the Java Runtime Environment at runtime. While this is not equivalent to the Java source code, Java bytecode is much easier to decompile than machine code (Hamilton, & Danicic, 2009). In this case, the binary was decompiled using a software tool called JD-GUI (a version of Java Decompiler) (Dupuy, 2017).



```java
/*
 * Main.java
 *
 * Created on 16 de mayo de 2006, 04:59 PM
 *
 * To change this template, choose Tools | Template Manager
 * and open the template in the editor.
 */

package inkarri;

import java.io.*;
import java.util.Random;
import java.util.concurrent.ConcurrentLinkedQueue;

/**
 *
 * @author arturodr
 */
public class Main {

    /** Creates a new instance of Main */
    public Main() {
    }

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        int tipo = 0; // 0, 1, 2, 3
        System.out.println("########################");
        System.out.println("Tema: "+Datos.temas[tipo]);
        System.out.println("########################");

        try{
            // Crea archivo
            archivo.inicializa();
            //baseDeDatos bd = new baseDeDatos();
//********************************* VARIABLES ******************************//
        System.out.print("Iniciando variables...");
        //ConcurrentLinkedQueue<String> buffer    = new ConcurrentLinkedQueue<String>();

        System.out.println("Conenctando a la base de datos...");
        baseDeDatos.conecta();

        Random numAleatorio     = new Random();
        int tLuegoDeImpresion   = 10000;
        int tNuevoProceso       = 3000;

        Random aleatorio = new Random();

        System.out.println("ok!");
//**************************************************************************//
```

```java
package inkarri;

import java.io.PrintStream;
import java.util.Random;

public class Main
{
  public static void main(String[] args)
  {
    int tipo = 0;
    System.out.println("########################");
    System.out.println("Tema: " + Datos.temas[tipo]);
    System.out.println("########################");
    try
    {
      archivo.inicializa();

      System.out.print("Iniciando variables...");

      System.out.println("Conenctando a la base de datos...");
      baseDeDatos.conecta();

      Random numAleatorio = new Random();
      int tLuegoDeImpresion = 10000;
      int tNuevoProceso = 3000;

      Random aleatorio = new Random();

      System.out.println("ok!");

      System.out.println("Llenando Base de datos...");
      for (int i = 0; i < 20; i++)
      {
        new parrafo(aleatorio.nextInt(2) + 1, tipo).start();
        try
        {
          Thread.sleep(5000L);
        }
        catch (InterruptedException ex)
        {
          ex.printStackTrace();
        }
      }
      for (; Thread.activeCount() > 2; ex.printStackTrace()) {
        try
```

Figure 10. Comparison of snippet of original Java source code (left) with decompiled code (right) for a binary files from Jose Carlos Martinat's *Brutalismo*. The decompiled code has been modified to include spaces where the header would be, to allow easier comparison with original source code.

Decompilation output (illustrated in Figure 10) again closely matches the original code, including project, package, class and variable names. As with *Becoming*, the primarily losses are the code comments and header metadata. While for *Becoming* this did not have a major impact on program comprehension, comments are potentially much more important in interpreting the *Brutalismo* software—a much larger project. However, the decompiled code retains a close resemblance to the original and would undoubtedly be valuable in developing program understanding in the absence of source code. In this instance, the decompiled code is also helpful in addressing an equivalence problem due to the proliferation of binary files on the host machine. Comparing the two allows a direct link between a component of the complex source project and an individual binary to be established. It is revealed through this process that the binaries only incorporate a subset of functionality contained in the source project. It should be noted that while bytecode decompilation was found to be very effective in this case—a conclusion which other evidence suggests might be widely applicable (Naeem, et al., 2007)—studies have also found Java decompilers (including the JD decompiler used in this case) to be unreliable in some cases (Hamilton, & Danicic, 2009). Java decompilation may be particularly difficult where code obfuscation techniques are used to counter it (Chan, & Yang, 2004), a technique which might be used to prevent reverse engineering of proprietary software.

For a work such as *Sow Farm* where source materials are not available for examination, decompilation could provide a means of filling this gap. However, decompiling this kind of large project, which was constructed in a graphical C/C++ based development tool called Quest3D, would be much more technically challenging than the prior examples. In this case, decompilation would need to target the machine code of the entire Quest3D engine in order to return a complete representation of the source code. In addition to the legal and ethical issues involved in doing so in this case (which I return to below), decompiling machine code is much more challenging than an intermediate representation such as Java bytecode. While decompilers targeting machine code do exist, the transformations that occur during the compilation process means that the results are typically much less useful and bear little resemblance to human-authored C or C++ source code (Jazdzewski, 2014). As a result, the decompiled program would require considerable effort and expertise to interpret, without any certainty as to whether all parts of the program are actually represented (i.e. the extent to which decompilation was successful). There is debate within the reverse engineering community as to whether decompilation of machine code into a complete high-level source code representation will ever be possible due

to the technical challenges involved (see Eilam, 2011 for a discussion of this). Even when taking the attitude that this is theoretically possible, until there are tools available that allows the process to be carried out reliably, the technique only has limited use in a conservation environment.

An alternative approach in this case would be to use a disassembler which can be applied to any machine code representation. A disassembler transforms machine code into a mnemonic representation designed to be more easily read by a human: assembly language (Geffner, 2014). As assembly language instructions have a one-to-one relationship with machine code instructions (Eilam, 2011), the volume of code produced (as well as the expertise required to interpret it) makes it considerably less useful when compared to source code or decompiled code. Despite the inherent challenges, the results of either decompilation or disassembly could—given enough resources put into their analysis by someone with the expertise—eventually allow reverse engineering of program understanding. The more pertinent question is whether this is actually worthwhile—the answer to which depends on the questions being asked. In the case of *Sow Farm*, much of the information required to plan an emulation-based preservation strategy could be gained through static binary analysis and other techniques which I will introduce in the next section. Accurately migrating *Sow Farm* to a new 3D engine on the other hand, would be very difficult without reverse engineering a more complete set of source materials and design documentation.

It is important to note that there are legal and ethical implications to the decompilation of proprietary software platforms—a prominent component of both *Becoming* (Flash) and *Sow Farm* (Quest3D). In UK copyright law, the right to decompile is, in certain circumstances, enshrined in law through a section of the Copyright, Designs and Patents Act 1988 (Atkins, 2009). While the use case highlighted in this section would likely count as an “acceptable objective”, Atkins found the Act to be unclearly defined. Similar legal ambiguities regarding decompilation exist in the United States (Behrens, & Levary, 1998). Exploring these issues in further detail is out of the scope of this thesis, but should be a consideration in the application of these techniques to software-based artworks which involve proprietary technology. Perhaps more important here are the ethical considerations. Gerrard has chosen to keep the source materials of *Sow Farm* in his care rather than pass them on to the museum—perhaps respecting this decision and working with the artist to alleviate preservation concerns offers the more appropriate pathway for this work.

4.5.2. Process Analysis and Instrumentation

While in the previous section I considered approaches which target software binaries as static digital objects; in this section I consider those which target the software as a *process*—that is, a program in execution (Silberschatz, et al., 2014). By directly addressing the binary program as an executing process, information may be gathered about the program's behaviour and performance, offering a potential alternative to directly examining code. The term *dynamic analysis* is used to refer to a set of methods which focus on the analysis of a software program while it is executing (Gosain, & Sharma, 2015). While this term is often used specifically in relation to methods that focus on debugging and testing code, here I adopt a broader definition that includes any method of intercepting or analysing software processes executing in a technical environment. The advantage of such techniques is that they offer precision and a goal-oriented strategy for understanding software programs (Cornelissen, et al., 2009). Dynamic analysis contrasts to *static analysis*, which focuses on analysing (source or binary) code as an object. Whereas static analysis can be used to exhaustively explore different executions scenarios, dynamic analysis is best used where a particular software characteristic or behaviour is targeted, and complete understanding of the system is not necessary (Stroulia, & Systä, 2002).

Approaches to dynamic analysis can be considered in relation to the point at which they intercept the software process in question. I will refer to the act of creating an interception mechanism (of any kind) as *instrumentation*. The most direct form of instrumentation is the addition of special lines of code to the source code which allow information to be captured when the software is executed. *Sow Farm* for example, permits monitoring of graphics performance and simulation data as the software runs, viewed through a hidden overlay feature (see Figure 11). This feature may never have even been intended for use by a collector or institution but may have been used to assist in testing and debugging the software during development.

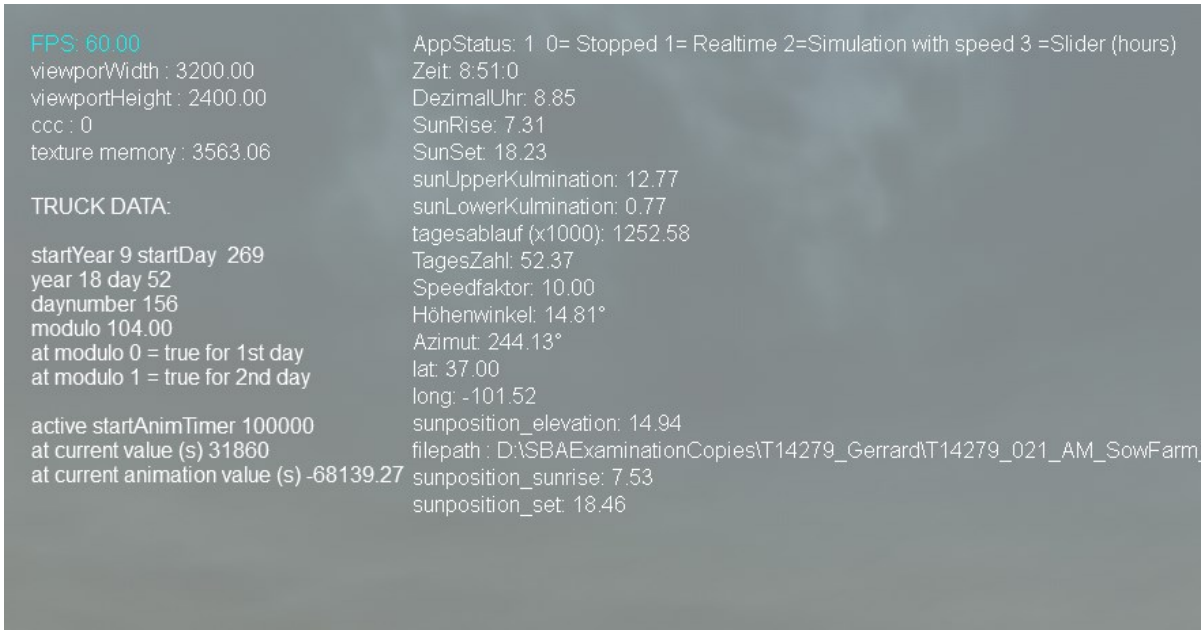


Figure 11. Screenshot of the debug overlay (which appears in the top left-hand corner of the rendered image), which is used for monitoring of *Sow Farm* while the software is running.

Becoming also contains instrumented code (using the ActionScript “trace” function), which allows the viewing of the live state of certain variables as the executable runs, if run through Abobe’s Flash Debug Player. In this case however, because of changes in the instrumentation requirements of Flash Debug Player, it is no longer possible to view the *Becoming* software in debug mode in current technical environments. This highlights one of the limitations of code instrumentation of this kind—it relies heavily on the nature of the instrumentation the artist (or collaborator) chose to hard code into the software, and (unless it is possible to revisit the code) it might be difficult to reliably maintain. Nonetheless, working with artists and programmers to implement or select appropriate instrumentation could be very beneficial for long-term preservation of software-based artworks by providing a means of verifying the accuracy of elements of a software performance—an issue I return to in Chapter 5.

Where code has not been instrumented (as in most cases where this has not been planned) and revisiting code is not possible, third-party dynamic analysis tools may be used to instrument the binary or intercept the process in some other way. Such tools can be built with a huge variety of goals in mind and can be implemented in many different ways. However, in experiments applying them during this research, I identified several generalisable method types which were particularly useful when analysing software-based artworks:

- **Profiling:** Designed to capture and log information about the performance of

a software program (or elements of its technical environment) as it is running. In a software-based art conservation context, this can be particularly useful in capturing and verifying software performance metrics such a rendering speed, execution times or hardware load. This might be useful, for example, in the verification of a software performance or in testing the software on a new system. Profiling some aspect of a software performance requires careful consideration of the appropriate metrics to use.

- **Tracing:** Designed to capture and log information about events and system interactions as a software program is running. In a software-based art conservation context, these techniques can be particularly useful in identifying calls to dependencies and other interactions with software environments. Operating system level tracing can be employed to reveal events such as file system interactions (which may indicate dependency), but this can produce very large quantities of data to be analysed. Program level tracing can be targeted more precisely and can give more detailed insight into program functionality, but may require negotiating machine code instrumentation, which brings with it the challenges of analysing machine code that were introduced in the previous section.

- **Data Monitoring:** Designed to capture and log data that is sent and received by a software program. Such tools could target a variety of communication protocols. For example, they might be used to monitor network activity (this is known as packet sniffing) or capture data being sent to a port (for example, to a printer or other hardware device). In a software-based art conservation context, this kind of information can be particularly useful for identifying the nature of a program's interaction with an external resource, or simply for assessing whether transmission is occurring.

Dynamic analysis techniques also have limitations with regards their use value. Stemming from their nature as goal-oriented strategies, the most significant of these limitations is that dynamic analysis is inherently an incomplete form of analysis, and only targets a portion of a potentially large and complex execution domain (Cornelissen, et al., 2009). Relying on such techniques for developing program understanding, in preference to source code analysis, therefore runs the risk of not capturing important elements of program function. Consider a hypothetical example, where a specific input triggers a program to make a dynamic call to a specific dependency. Unless this specific input were triggered (and knowing how to trigger it

might itself require in-depth knowledge of the program), this dependency may never be captured by dynamic analysis methods. While any information is valuable in cases where source materials are unavailable, we might be cautious when considering whether to base more significant preservation actions (for example, reimplementing a program in another programming language) on information gathered through dynamic analysis.

Ultimately, dynamic binary analysis tools are complementary to other approaches such as static binary analysis and source code analysis, and can be used alongside them in cases where this is possible. In the next section I describe a case in which both dynamic and static binary analysis are applied to answer questions about a software program. However, for those cases where the utility of source code-centric analysis is constrained in some way, dynamic analysis tools provide another means to gather information about a software program. Performance verification, an issue I return to in Chapter 5, may be where dynamic analysis will be most useful for analysing software-based artworks. In these cases, working with artists to build or specify appropriate software instrumentation may be particularly valuable.

4.5.3. Case Study: Dependency Identification Using Binary and Process Analysis

Identification of the technical environment required to run John Gerrard's *Sow Farm* is an important task in the examination and documentation of the work. This is made particularly important by the existing interest in virtualisation as a preservation strategy for this work (Falcão and Dekker, 2015)—migrating this work is not an option—which would require an understanding of how the technical environment is constructed. Doing this is challenging however, as the software relies on a complex 3D rendering pipeline. For this software, which was designed for the Microsoft Windows operating system (OS), the primary interface between the software and the graphics hardware is the DirectX API. DirectX has been under development in some form since the mid-1990s, and has had a number of core versions roughly paralleling the history of the Windows operating system. These core versions (the most recent at the time of writing is DirectX 12, which ships with Windows 10) have offered an evolving feature set. As new versions are released, older functionality is sometimes deprecated and even phased out.

While a version of the DirectX runtime (the component required to run software developed for DirectX) is included with all versions of the operating system family since Windows 98, compatibility of contemporary versions of Windows with

applications written for older DirectX versions varies as a result of the gradual changes to the API. To combat this problem, Microsoft makes granularly versioned runtime libraries available, in order to provide backwards compatibility for older applications. This is not an unusual approach for backwards compatibility among runtime libraries, but results in a proliferation of versions. The version used by a particular software program will depend on the version of the DirectX SDK used during its development. This can be quite specific, new versions having been released as frequently as monthly during some periods. For this reason, the installation of an additional runtime library is sometimes necessary in order to run a program. Where there is no well-defined installation process, as in the case of *Sow Farm*, it is important to identify which versions of the DirectX runtime libraries the software requires.

With no source materials available for study, unambiguously identifying these dependencies might fall to other methods of binary-centric analysis. An initial problem encountered was that analysing the Windows Portable Executable from which the software was launched using CFF Explorer (Pistelli, 2012) does not return information about Dynamic Link Library (DLL) dependencies—the kind of dependency that the program has in relation to DirectX runtime libraries. This suggests something is happening when the program is executed that results in this information being hidden, so we might instead consider addressing the running process instead to reveal what. Using Microsoft Sysinternals Process Monitor (ProcMon) tool (Russinovich, 2017) to carry out a system trace analysis, it is possible to log all the file read and write operations being made as the software was executed, generating a very large quantity of data.

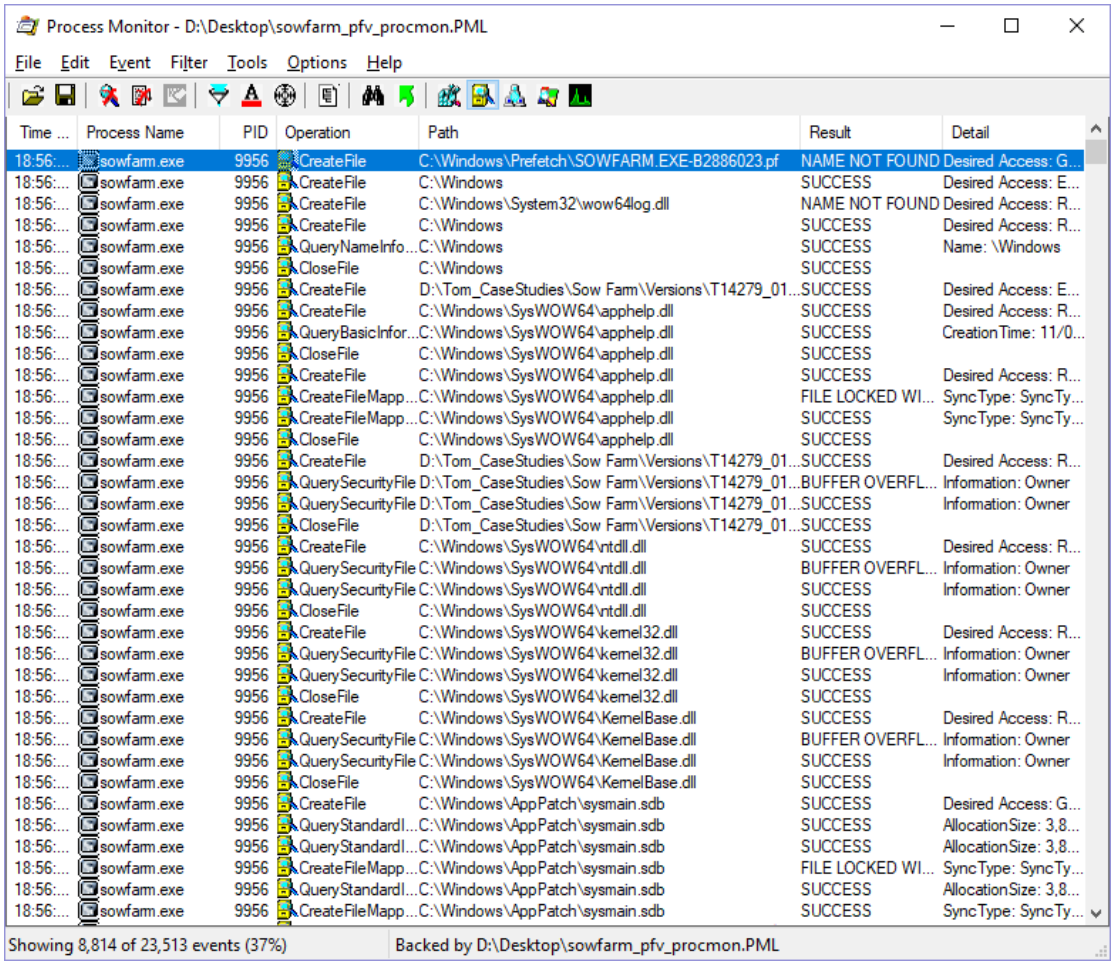


Figure 12. Screenshot of the Sysinternals Process Monitor program (Russinovich, 2017), showing file system activity logging results for the sowfarm.exe software process. Each line represents a file system activity.

In this case, carefully examining the log data from the sowfarm.exe process file system trace reveals that the software was unpacking the contents of the executable to a temporary directory behind the scenes and executing an unpacked program from there. With this knowledge, it is possible to make a copy of this data while the process is running (it would normally be deleted when the process was terminated) and examine this extracted data in detail.

With the correct executable representation of the software identified, we can now again attempt to use binary analysis to derive information about its dependencies. However, the correct binary to address is unclear: there are 195 files in the extracted directory, many of which are DLL files which could pose their own dependencies. While it is possible to analyse these one by one, a more effective approach is to, again, analyse the process directly at runtime. Using ProcMon on the process, we find that a set of DLL files with d3d9 or d3dx9 in their file name are being loaded, the

naming of which indicates that they are DirectX 9 related runtime libraries:

Time	Process Name	Operation	Path	Result
56:23.6	QuestViewer.exe	CreateFile	C:\Windows\SysWOW64\d3dx9_25.dll	SUCCESS
56:23.6	QuestViewer.exe	CreateFile	C:\Windows\SysWOW64\d3d9.dll	SUCCESS
56:23.7	QuestViewer.exe	CreateFile	C:\Windows\SysWOW64\d3dx9_36.dll	SUCCESS

Table 4. DirectX library read results of a trace analysis of QuestViewer.exe process using Microsoft Sysinternals Process Monitor (output to a CSV file and edited here for clarity).

The first and third entries in Table 4 are both runtime libraries (the middle entry being the core library), and are versioned with the numbers at the end of their file names: 25 and 36 respectively. This allows us to find the appropriate runtime library installer package distributed by Microsoft. This is a useful starting point for disentangling the web of dependency relationships posed by the *Sow Farm* executable that spread into the technical environment within which it is embedded, and the steps could be repeated to identify other dependencies of different kinds. This is important because when this software is emulated or installed on a new host machine for display, we need to be able to reconstruct an appropriate execution environment from scratch.

4.6. Representing and Describing Software Structures

For the last part of this chapter I shift focus from analysis to *representation* of the results of analysis. This is a crucial stage in the workflow introduced in Section 4.2, which ensures that knowledge derived from reconstructive analysis is captured in a form that can be used to inform conservation activities. There is some overlap here with the metadata requirements for the representation of software-based artworks in information systems, which must also represent the elements that constitute the basis of a software performance and their relationships with the work's versions and realisations. While it is important to capture the analysis stage itself, in terms of both the resulting data and the descriptive narratives of the process (examples of these are found in the case study sections of this chapter), these are highly dependent on the kinds of analysis and tools used. The structured metadata representation of the constituents of a software performance, on the other hand, can be considered, to some extent, independent of how the information was derived.

A structured metadata representation may also have potential in serving as a high

level architectural overview of the components of a software system, an artefact valued by software engineers (Das, et al., 2007, Lethbridge, et al., 2003, Tilley, et al., 1992). The Institute of Electrical and Electronics Engineers (IEEE), one of the key bodies in the standardisation of software engineering practice, defines architecture as comprising the “fundamental concepts or properties of a system in its environment embodied in its elements, relationships, and in the principles of its design and evolution” (anon. ISO/IEC/IEEE Systems and software engineering – Architecture description, 2011, p.2). An overview of architecture for a legacy system might include the components of the system, the external interfaces with its environment, and the relationships between them (Hilton, 2016). Given that collaboration with software specialists may be required in the long-term care of software-based artworks, a high-level representation of the software architecture may be of value in communication between parties.

As discussed in Chapter 3, the value of a representation might be best understood through the extent to which it is useful to those using it. In considering how to direct the formulation of an appropriate representation, we can return to the roles of information systems in conservation activities identified in Chapter 3:

- Management of physical and digital objects, including tracking of their locations and recording of loans, and their relationships with an artwork and its realisations through time.

- Serving information to support analysis of and reporting on the characteristics of the collection or a subset of the collection.

- Allowing computer systems to manage and manipulate digital objects stored in a repository.

These roles are somewhat generic and could apply to any time-based media artwork. In Chapter 2 I discussed the distinction between the realisation of a software-based artwork and the software performance that occurs within this realisation—in this section I am only considering the latter. With this in mind, we can further refine the potential uses a representation of a software structure might have for a conservator:

- Providing information about the discrete, locatable hardware and software components that were used to achieve a software performance, where they are located, and how they relate to each other.

- Making clear particular characteristics of a performance, such as whether processes of technical abstraction (e.g. emulation and virtualisation) were employed or whether external resources (i.e. those which cannot be acquired as digital materials for preservation) are required.

- Identifying how many software-based artworks employ a particular hardware or software component (e.g. that require a Mac OS X operating system or that were developed for the Flash platform) for achieving a particular software performance.

- Providing a means for a digital repository system to serve appropriate digital resources (i.e. the components required to prepare a particular software performance) when required for exhibition or display.

This list provides a baseline from which we might judge the suitability of a representation of a software structure and the metadata needed to populate it. In the following sections I consider the extent to which existing approaches to metadata and modelling (as identified in Section 3.3.3) might be used to describe software structures in a way which can fulfil these uses effectively.

4.6.1. Appraising Existing Standards and Models

Richard Rinehart's **Media Art Notation System** (MANS) approach aimed to create a structured method for "scoring" an artwork, which could "constitute a guide to aid in the re-creation or re-performance of the work" (Rinehart, 2007, p.183). This model's basis in ideas of performance is attractive for our requirements, so warrants further analysis. The descriptive elements of the model offer limited value for describing qualities specific to software-based artworks, as they simply map to Dublin Core elements, the dominant standard for collections metadata, which the majority of collections management systems already support. The structural elements of MANS are more novel, as they present a conceptual model for media artworks. This model consists of an "artwork" which is made up of "versions" (similar to our 'realisation'), which are made up of "parts" (or components), which in turn consist of "resources" (physical or digital things). This bears a close resemblance to our understanding of time-based media artworks, and so provides a useful high-level model. However, it does not incorporate sufficient structural complexity to allow the requirements of the software performance to be modelled at a lower level: modelling the software as a "resource" would ignore the complexities of the software super-object and its technical environment completely.

PREMIS is the *de facto* preservation metadata standard for digital objects, its primary application being in the management of digital objects and associated preservation activities (PREMIS Editorial Committee, & others, 2015). PREMIS can be integrated with other metadata schemas through the use of specific identifiers applied to objects (typically files), events or agents, and operates primarily at the file (or package of files) level. An essential tool in implementing high quality preservation metadata, PREMIS will be an equally important standard for software-based art. However, the diffuse nature of software-based art (i.e. the connectivity between software super-object and technical environment), does raise some issues with PREMIS' focus on digital objects. Version 3.0 of the standard introduced support for the capture of "environments", which are modelled as objects in themselves and linked to the associated digital object by a dependency relationship (Dappert, et al., 2016). PREMIS also models the purpose of an environment in relation to an object (a classification of "create", "render" and "edit" is available for use) and the extent to which an environment supports that object ("minimal", "recommended" or "known to work"). Environments are composed of other entities, which might in turn be composed of still other entities—so allowing the construction of a representation of a complete environment down to the level of granularity at which it will be managed. While the terminology remains somewhat unrefined and its application untested in relation to software-based art, PREMIS 3.0 appears to present a set of modelling options which would capture the fundamental components of a software structure. However, it lacks the descriptive detail through which a representation of sufficient detail (in order to support the uses outlined earlier in this section) could be constructed.

A similarly granular approach emerged from research in digital preservation more than a decade prior to this: the **Capturing Unstable Media Conceptual Model** (CMCM), an ontology developed by the V2_ organisation's Capturing Unstable Media project (V2_Institute for the Unstable Media, 2003, V2_Institute for the Unstable Media, 2003). CMCM provides a structure for the "capture" of an artwork or occurrence as a specific event in time and for its explicit linking to associated documentation. The CMCM is not designed to provide structure to a database or to be implemented as an out-of-the-box solution, but rather, "may function as an independent reference framework" (V2_Institute for the Unstable Media, 2003, p.15)—in essence it is a conceptual model. While the modelling choices made in the construction of the ontology are not completely clear from the project documentation, an examination of the published ontology reveals that specific consideration has been

given to software as a type (or component) of a “captured thing”. This includes capture of elements of a technical environment, including a form of dependency linkage through relationship assertions between software “applications” and “configuration” (a grouping entity for other components) instances. Due to the broad scope of the model, the level of detail that can be captured is rather limited in terms of the explicit modelling of software and hardware environments, and there are only a limited set of entity types defined for the constituents of these environments. Nonetheless, CMCM represents a significant contribution to the challenge of modelling complex time-based media artworks and provides a valuable starting point for a model of software performances.

The EU FP7 **PERICLES**[17] research project, which ran between 2013 and 2017, developed a model-driven approach to the preservation of complex digital objects. The outputs of the project include a set of digital preservation ontologies, designed primarily to model digital resources within a changing technical environment or “ecosystem” (Waddington, et al., 2016). Unlike PREMIS and CMCM, the approach taken is somewhat modular, in that a wide array of digital object types might be modelled at their respective domain level and connected using an upper level ontology called the Linked Resource Model (LRM) (PERICLES Consortium, & others, 2014). The LRM has its roots in the PROV-O ontology, one of the W3C standards for exchange of provenance information over the web—a relationship it shares with PREMIS. The LRM is similar to PREMIS in its digital preservation purview, with a slightly different degree of specialism, primarily to model complex dependencies between digital objects in an operational environment. While the LRM is too generic to model software structures, an ontology design pattern for computer systems also resulted from the PERICLES project (Mitzias, et al., 2017)—this may have relevance given that it was produced in relation to work in the software-based art preservation domain. This pattern models the software and hardware components that make up a computer system, and the dependencies and compatibility between them. While the model is rather simplistic (only five classes of entity are defined), its modelling of dependency is a useful conceptual foundation for further work. Particularly interesting is the decision to model dependency through two types of relationship: “uses” and “requires”, which respectively indicate a soft (should be maintained) or hard (must be

[17] For more information about PERICLES, see the project website: http://www.pericles-project.eu/

maintained) dependency. This distinction is likely to be arbitrary in some cases, as it can be practically difficult to determine whether a dependency is of one or the other type.

CIDOC-CRM is a conceptual model for enabling interoperability of museum information systems (Le Boeuf, et al., 2015). It does not, therefore, attempt to specify the precise nature of any underlying data structures, but rather presents a high-level model which enables mapping between systems and approaches. By design CIDOC-CRM does not model to a level of detail that would allow capture of the relationships between the technical components of a software-based art system. Of more interest in this regard is its digital extension, CRMdig, which has been applied to the description of time-based media artworks by Juergen Enge and Tabea Lurk (Enge, & Lurk, 2014). This is an interesting approach which captures the performative nature of such artworks well in the examples developed, which include an internet artwork, by using the event modelling components of CRMdig. The artwork in this case is not modelled as something consisting of components, but rather it is the output of a "digital machine event" which draws upon data inputs to yield the digital object as it is experienced. Missing from this approach, in the examples given, is any modelling of the software super-object as a concrete digital thing (which in all the case studies I have examined, it is) or of the relationships between artwork, realisation and components. This makes it unsuitable for use in the management of concrete digital objects.

Among the approaches I have examined in this section, none offers a fully realised approach to the structured representation and description of software-based artworks when considered in isolation. Furthermore, there are few case studies from the software-based art conservation domain to demonstrate their value. If metadata is to be placed into effective service in the conservation of software-based artworks, there is the need for a clear conceptual model of what the software structures must capture, grounded in the realities of managing a set of physical and digital components. There is however, evidence of sufficient potential in existing models to allow them to be integrated usefully with such a conceptual model and so maintain links with relevant standards—particularly in the case of PREMIS, which is widely used in the digital preservation domain.

4.6.2. High-Level Perspectives on Software Structures in UML

Now faced with the challenge of defining an appropriate conceptual model for

representing software structures, we might look again to the approaches employed in the established field of software engineering. The ubiquitous representational language in this field is Unified Modeling Language (UML). Its *de facto* maintainers, the Object Management Group standards consortium, state that UML is designed to help, “specify, visualize, and document models of software systems, including their structure and design” (Object Management Group, 2005). UML is a flexible language and can be used to represent diverse software structures at different levels of abstraction. While, unlike the standards and models examined in the previous section, it is not designed for knowledge organisation, it may offer principles from which we might draw.

The use of UML in the context of software-based artwork source code documentation has been explored by Deena Engel in collaboration with Glenn Wharton (Engel, & Wharton, 2015) and Mark Hellar (Engel, & Hellar, 2014) in research on museum collections. It is suggested that UML may “give future programmers an overview of the system as a whole, and how different aspects of the software work together” (Engel, & Wharton, 2015, p.94). These studies focus on producing class diagrams (a UML subset for the representation of object-oriented programming structures) for source code, however, which is unsuitable for describing the higher level of abstraction which has been the focus of this chapter. Other parts of UML operate at this higher level. The deployed (that is, put into use) hardware and software components of a system are best represented using the language’s *deployment diagram* type. In the following discussion and examples, I refer to and use UML Version 2.5, the most recent version of OMG standardised UML (Object Management Group, 2015). This version defines the deployment package as specifying “constructs that can be used to define the execution architecture of systems and the assignment of software artifacts to system elements” (p.651). A deployment diagram uses *nodes* and *artifacts* to represent the concrete components of a system. Nodes typically represent hardware devices or software execution environments, and may nest within each other (e.g. an operating system execution environment nests within a computer system). Connection pathways can be made between nodes to indicate flow of information between devices or execution environments. Artifacts represent the products of development such as executables, scripts, libraries and databases.

In Figure 13 below, the software system associated with artwork *Brutalism* (as realised in 2011 at Tate) by Jose Carlos Martinat Mendoza is represented as a UML deployment diagram. This model was constructed based information gathered from

an interview with the developer of the *Brutalism* software, Arturo Diaz Rosemberg (carried out by Patricia Falcão at Tate), and my own examination of the source code, binaries and software environment. The examination was carried out within a virtual machine using a captured disk image (see Section 4.2 for a description of this workflow).

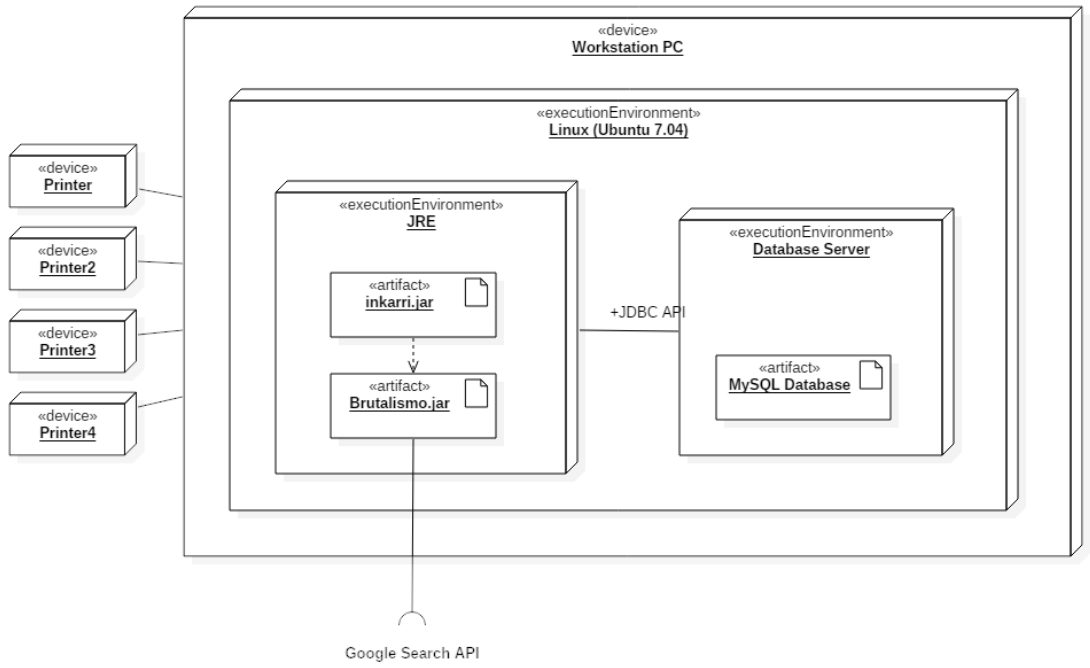


Figure 13. The hardware and software components of the 2011 realisation of *Brutalismo* represented as a UML deployment diagram. 3D boxes are nodes, boxes with file symbols in their top right-hand corners are artifacts, solid lines indicate (non-directional) communication pathways, dotted arrows indicate dependency relationships, while semi circles indicate external interfaces.

Although the diagram elements require a level of specialised knowledge to decode, this UML deployment diagram succinctly conveys a considerable amount of information about the structure of the system it represents. Digital objects are identified clearly using the artifact type, while their relationship with their technical environment is indicated through the use nested layers of nodes. We can easily determine that emulation or virtualisation are not being used in this realisation, as the primary execution environment is nested within a device node. The fact that Java Runtime Environment (JRE) and MySQL are required in order to use particular binaries is implied through nesting. We can also determine that there is a technical interface between the JRE and the database (and its type, JBDC), and between one of the binaries and the external Google Search API.

Despite its value as a means of effectively visualising a complex system, the UML deployment diagram is of limited value in terms of integration with information systems—the primary use case for structured representations of software structures. This is because, as a modelling language designed primarily to produce diagrams, UML lacks the ability to encode formal semantics or data properties that could describe artwork components in detail. For example, if we want to be able to query how many artworks within a collection involve peripherals with RS-232 connection interfaces, this would be impossible to reason using a UML model defined at the deployment level, which would only indicate a communication path between one named hardware device and another. Furthermore, UML, as a maintained standard, does not accommodate the extension of its principles with domain-specific knowledge. This ability to extend beyond the software engineering focus of UML is important, as to make sense of descriptions of software-based artworks we need to be able to relate software structures to the various version, variants and realisations of artworks.

An ontology-based approach to modelling is proposed as a more appropriate solution. Ontologies (as introduced in Section 3.3.3) are systems of knowledge representation which include provision for formal semantics and are designed to explicitly accommodate the specification of domain knowledge (Munir, & Sheraz Anjum, 2018). Nonetheless, the successful elements of the UML deployment diagram identified above have implications for how an ontology-based conceptual model should be specified. Firstly, execution environments must be explicitly modelled (and distinguished from the software required to create them) and related to one another in order to capture deployment requirements. Secondly, relationships between software components must also be explicitly modelled to indicate that a technical interface is required between components.

4.6.3. Conceptual Model for Representing Software and Environment

In this section I will briefly introduce a conceptual model developed in response to the challenges discussed in earlier sections. This model was designed to capture representations of realisations of software-based artworks by describing their software and hardware constituents, the properties of these constituents and the ways in which they relate to each other. Model elements were develop iteratively based on insights gained from the close examination of the technologies employed in the artwork case studies, using the methods of software analysis described earlier in this chapter. Three case studies of varying levels of technical complexity were

modelled using the ontology developed—*Becoming*, *Sow Farm* and *Brutalismo*. The model is specified as a Web Ontology Language (OWL) 2 ontology (World Wide Web Consortium, 2012), the *de facto* standard for ontology for authoring ontologies. However, the conceptual model it represents was designed to be technology-agnostic with regards implementation and functions as a standalone model for guiding the description of software structures.

The model is intended to provide a structured representation both as a form of architectural overview and description, and as a preservation information resource. Although designed in the context of describing software-based artworks, the ontology could describe other structures where the software performance model is relevant. A brief summary of the model and a use case example are presented below. In Appendix II, the complete set of classes and properties that constitute the model are specified in detail, including a description of each element. The model is titled the 'Software-based Artwork Structure Ontology' and is presented as an RDF/XML format OWL 2 (World Wide Web Consortium, 2012) ontology developed in protégé 5.2 (Stanford Center for Biomedical Informatics Research, 2016). The ontology is also available for re-use under a Creative Commons BY-SA 4.0 licence via GitHub (Ensom, 2018).

Focusing on the realisation of an artwork in time and space, the realisation is modelled as being constituted of several key entity types which map to PREMIS 3.0 semantic units (PREMIS Editorial Committee, & others, 2015):

- **Hardware Environment** (maps to PREMIS 3.0 *Intellectual Entity* of type *environment*): the hardware portion of a technical environment in which software can be executed;
- **Software Environment** (maps to PREMIS 3.0 *Intellectual Entity* of type *environment*): the software portion of a technical environment in which software can be executed;
- **Software Super-Object** (maps to PREMIS 3.0 *Intellectual Entity*): the set of digital objects which constitute a unique expression of the software.

Using PREMIS, the Intellectual Entities could be linked to relevant Representations (e.g. a raw disk image capturing a software environment). Hardware Environment and Software Environment entities are linked to the Software Environment entities they support using the hostsEnvironment object property. Software Super-Object entities

are linked to suitable Technical Environments using the isExecutableIn object property.

Hardware Environment, Software Environment and Software Super-Object entities can each be composed of:

- **Hardware** (maps to PREMIS 3.0 *Intellectual Entity*): a hardware component;
- **Software** (maps to PREMIS 3.0 *Intellectual Entity* or *File*): a software component;
- **Data** (maps to PREMIS 3.0 *File*): a data component.

For each of these, relationships can be indicated by the hasHardwareComponent, hasSoftwareComponent and hasDataComponent object properties (detailed usage restrictions are specified in the full model). For each of these a preliminary set of types is proposed based on the software-based artwork case studies examined. Although PREMIS 3.0 does include a vocabulary with similar coverage (for the *environmentFunctionType* property), this currently conflates technical environments and discrete software/hardware components, which limits is usefulness in this context.

It should be noted that this model does not explicitly model dependency. This was found to be unnecessary, as all dependencies are inferable through the modelled relationships between software program and execution environment. Rather than take the approach of the PERICLES software system domain model and explicitly model them as relationships of “requires” or “uses” (Mitzias, et al., 2017), assertions which are difficult to make with certainty in practice, I propose that a better approach is to consider dependency in relation to environments that are known to have been used to achieve a software performance. A dependency is therefore inferred from a Software Super-Object to the constituents of those environments in which it has been executed when the artwork has been realised in the past (isolating which are essential requires the use of reconstructive analysis, as described in Section 4.2). The approach developed also ignores configuration issues—that is, the user definable parameters of a particular component—as modelling attempts found that these were too variable and complex to be captured in a way that would make them useful within a collections management system or digital repository. There is in any case, relatively little value to be gained from describing configuration in an information system, as it is not applicable in the tracking of physical and digital components, but more

frequently considered when a work is realised. The association of clear identifiers with each component within a technical environment would allow relationships between components to be established with other documents, which themselves offer a suitable description of configuration requirements.

The application of the set of classes and object properties defined is demonstrated in Figure 14 below. This diagram represents the modelled constituents of the 2011 Tate Modern realisation of the *Brutalismo* artwork case study as expressed in the OWL ontology developed. This model incorporates formal semantics, which ensure that the properties of individual components and the relationships between them are captured. For example, the Software components that constitute a Software Environment are modelled using the hasComponent property, which in turn makes it possible to reason that the Software Super-Object (connected to the Software Environment by the isExecutableIn property) has a dependency on those Software components. While this formal expression in a machine-readable language means the model is well suited to integration with information systems, it also has disadvantages. For example, it is harder to achieve the clarity of visual representation achieved in the earlier UML deployment diagram, which uses a defined notation to convey information. However, as semantics are encoded into the model there is the possibility of generating a UML diagram from the OWL ontology, providing tools are developed to carry out this transformation.

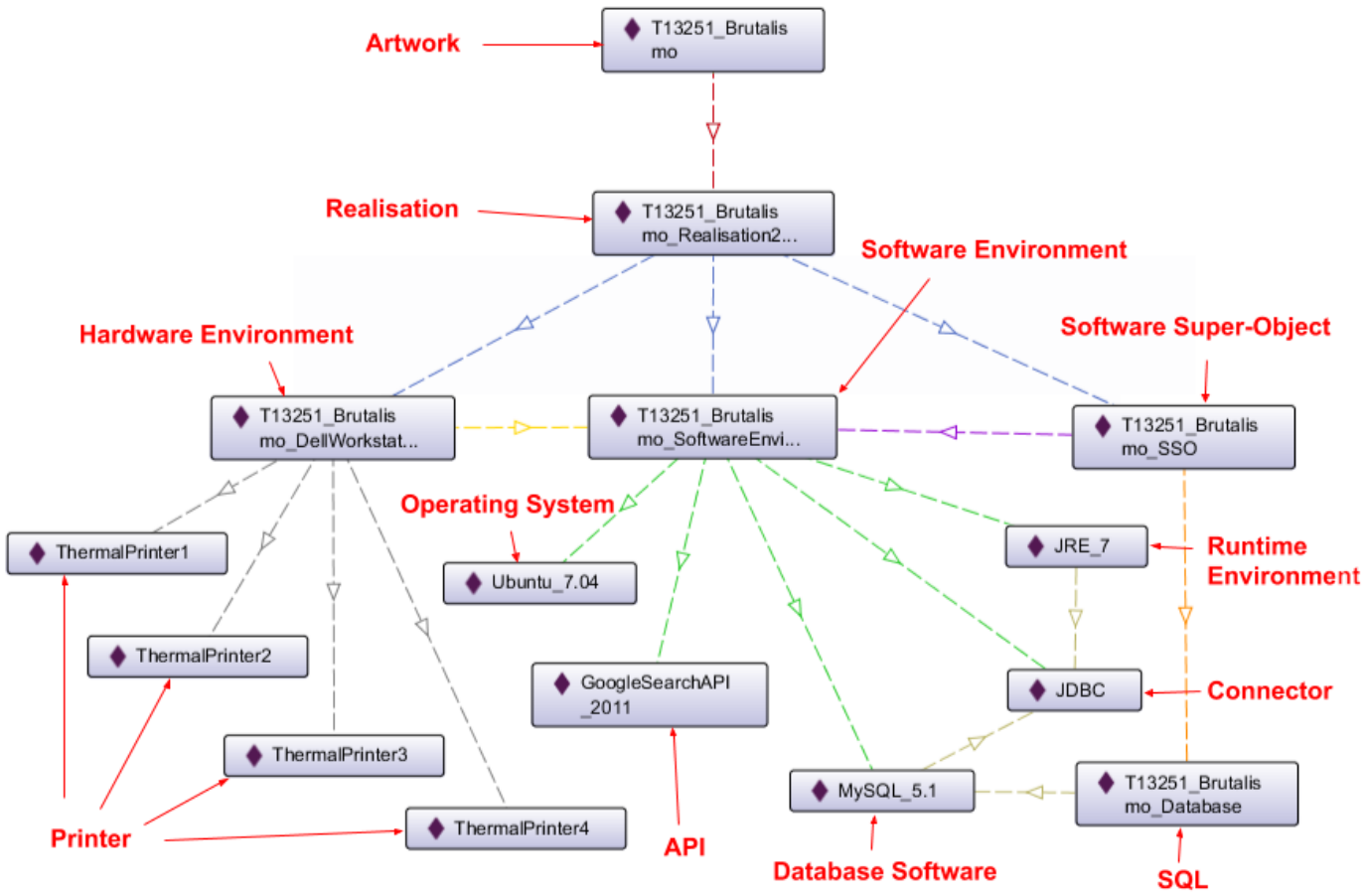


Figure 14. Representation of modelled entities for the 2011 realisation of *Brutalism*, produced using the Protégé 5.2 OntoGraf plugin. Boxes represent instances, red labels indicate classes, while object properties are represented by colour coded dashed arrows (red: hasRealisation; blue: hasConstituent; yellow: hostsEnvironment; purple: isExecutableIn; grey: hasComponent; green: hasSoftwareComponent; brown: hasInterface; orange: hasDataComponent)

4.7. Chapter Summary

In this chapter I have presented a pragmatic approach to the analysis and representation of software structures for use in the conservation of software-based art. This approach was developed in the context of the processes of examination, analysis and reporting demanded of the conservator in the long-term care of such artworks, as well as the desire to create contained, generalised representations of software and its technical environment through the process of reconstructive analysis. Conservation strategies which seek to keep software systems running have similar goals to those of software maintenance, while software-based artworks may have parallels with poorly documented legacy systems. These links allows the effective repurposing of existing approaches to analysis from the discipline of software engineering. Reverse engineering becomes the most appropriate way of deriving program understanding from a legacy system, ultimately providing a means for the conservator to create effective documentation that stands in for detailed development documentation where this is absent or in some way limited.

While source code centric strategies have dominated discourse in the area of software-based artwork analysis, in this chapter I problematised this approach and offered a set of complementary approaches for the interrogation of software structures. These approaches navigate the liminal materiality of software in order to reveal hidden information about the software representation they address. Binary analysis can be used to unpack and interrogate the opaque, machine-oriented representations of software which form the executable software components of a software performance. Process analysis can be used to interrogate the software as a computation process rather, and so intercept the actions of the system as a software performance occurs. While suffering from their own respective limitations, these approaches provide valuable tools for the software-based art conservators toolbox that both complement and offer an alternative to source code analysis. They are likely to be particularly effective where a preservation approach is taken which aims to maintain access to software through maintenance of an appropriate technical environment, as the software super-object is likely to remain largely unaltered in these cases. While these approaches may also provide insight into the functionality and implementation of the software, these insights are often limited by the extent to which code and process at the machine level can be comprehended in practice by a human reader.

Insight gained from these analysis approaches may be particularly significant in

capturing information that describes a particular software performance, which can then be used to create a representation of the underlying structure for storage within an appropriate information system. Such a representation benefits not only the management of digital objects within a collection, but ensures that information about a particular performance is documented for use in the display and study of that work in the future. I propose that an effective way to capture this information is through the use of a well-defined model of the technical environment in which a software super-object was performed, and the hardware, software and data components that constitute this environment. This is useful not only as a means of structuring machine-actionable metadata records in the service of conservation, but as a tool for representing artwork realisations and supporting understanding of system architecture. The model I have presented is a domain ontology (written in OWL 2), which maps to the core components of the dominant digital preservation metadata model PREMIS 3.0, while offering further clarity over the semantics of a software program's relationship with its technical environment. While the model is expressed in a machine-readable language and might be implemented as-is, it may be most useful as a tool for guiding the extension of existing systems of structured representation to better support software-based artworks.

CHAPTER 5

SIGNIFICANCE AND IDENTITY IN THE SOFTWARE PERFORMANCE

5.1. Chapter Outline

In Chapter 3 I suggested that the identity of a software-based artwork might be understood in relation to a set of significant properties, the maintenance of which helps to ensure that future realisations are authentic. However, precisely how significant properties might be used to represent identity in practice is unclear. Maintaining identity is likely to be a particularly important consideration where change occurs due to the loss or obsolescence of specific components and where there are shifts in the context of the work. In this chapter I will identify how existing frameworks for the capture of this kind of information might be applied to software-based artworks and, where they are found insufficient for this purpose, how they might be extended.

I start the chapter by revisiting the notion of significant properties from the digital preservation domain (including its relationship with conservation theory) and critically considering the value of using such a framework in the conservation of software-based art. A particular theoretical concern is the practicality of identifying significance among large numbers of variables, particularly in relation the complex set of material

considerations posed by the software medium. Other challenges concern how properties might extend beyond the object of conservation, particularly in relation to the variable nature of the connection between of the meaning attached to materials used. Maintaining a focus on practical solutions, in the second half of the chapter I develop frameworks for identifying significance at the level of the software performance.

5.2. Significant Properties and Identity

A recurring idea in the preceding chapters is that software-based artworks change over time. The circumstances of their realisation, the specific components and technology, and the social and technological contextual of the work, may all vary between realisations. For conservators, it is therefore important to understand what the acceptable parameters of change are and ensure that a software-based artwork can still be realised as an authentic representation of the artwork's identity. Even where change is occurring slowly or is not permissible at all, the conservator needs documentation to ensure that the particular realisation can be verified as acceptable. As I noted in Chapter 3, these are not new ideas and have found currency in both digital preservation (significant properties) and time-based media conservation theory (work-defining properties). There is however, a noticeable gap between theory and practice. A lack of published methodologies for the identification, capture and verification of the properties that constitute an artwork's identity—and few examples of these principles being used in the real world—raise questions over their value. In this section I revisit the concept of significant properties and related notions, and consider their potential use in caring for software-based artworks.

5.2.1. Revisiting Significant Properties

Significant properties, sometimes used interchangeably with *significant characteristics*, is a widely used but vaguely defined concept in the field of digital preservation. The concept's origins and the ambiguities over its definition have already received some critical attention (Dappert, & Farquhar, 2009; Giaretta, et al., 2009) so I will not repeat this work here, but rather consider their suitability for the specific use case of software-based art conservation. The definition developed during the significant properties focused research project InSPECT is one of the more widely cited definitions, and remains representative of a general understanding of the term:

> "The characteristics of digital objects that must be preserved over time in order to ensure the continued accessibility, usability, and meaning of the objects, and their

> capacity to be accepted as evidence of what they purport to record". (Wilson, 2007, p.80)

I will briefly examine some of the terminology used, analogues in conservation theory, and challenges posed to this definition by software-based art. The first and most obvious consideration is that, when we look at conserving software-based art, we are not dealing with discrete “digital objects”. As illustrated in Chapter 2, software-based artworks are structurally complex and their digital object components tend to be numerous (hence the definition of the software super-object as a grouping concept), interlinked and at times highly dependent on the technical environment in which they are situated. The software is experienced as a performance, and as such, the boundaries of the work as any identifiable digital thing are often unclear. This is problematic for applying the significant properties concept, as potential properties might have to be identified at multiple levels—the digital objects, the software and hardware environment, and the performance itself—which are closely linked.

In the InSPECT definition, “accessibility, usability and meaning”, are supplied as the motivations behind significant property preservation. While this kind of terminology might not be typical in art conservation, the concerns they reference actually align well. Conservators too, are concerned with continuing “access” to and “usability” of artworks—most crucially evidenced by their display—and with ensuring that their “meaning” is maintained in the process. The latter part of the definition, and the notion of “evidence” in particular, is again not typical terminology in a conservation context, yet there are clear analogues. “Evidence” seems to align closely with the idea of *authenticity*, the navigation of which in relation to artistic intent is a recurring topic of interest (and debate) in the conservation field (Laurenson, 2006, Hermens, & Fiske, 2009, Scott, 2015). In the same way that archival records must be accepted as “evidence of what they purport to record”, software-based artworks should be accepted as evidence of the identity of the work, authentically realised.

As we might expect given this alignment of concerns, analogous theoretical frameworks have emerged in the time-based media art conservation field. In Section 2.3 I discussed Goodman’s autographic-allographic distinction as applied by Pip Laurenson to the conservation of time-based media artworks. Laurenson proposes that the identity of time-based media artworks can be understood as a “cluster of work-defining properties” (Laurenson, 2006). Elements of the concept and terminology used clearly align with significant properties. There are also some subtle differences between the digital preservation and conservation perspectives on

significance, however. Applying the theories of philosopher Stephen Davies, Laurenson introduces the idea that an artist may specify the properties of an artwork “thinly” or “thickly”; the former being very precisely specified and the latter allowing for a degree of variation between realisations of the work (Laurenson, 2006). Prevalent notions of significant properties do not typically incorporate the same flexibility perhaps due to their focus on the aforementioned digital object, which stands in contrast to the more explicit acknowledgement of the performative qualities of the object of conservation in the field of time-based media art conservation.

A second key difference is the privileging of the artist’s authorisation in the realisation of time-based media artworks. While author and intent are still relevant in digital preservation, in areas such as data archiving and libraries more focus might be placed on the requirements of the users of the digital materials in question. The user in a conservation context (e.g. gallery or website visitor) on the other hand, is rather more passive in relation to the process of defining significant properties. The question of the extent to which the desires of the artist should be prioritised over other concerns is in itself a challenging issue in art conservation (Gordon, & Hermens, 2013, Wharton, 2016). Any in-depth examination of these issues is beyond the scope of this thesis, but it is important to note caution in relying on any single account of the artist’s perspective on the intentions behind their work. While such accounts are undoubtedly important, in some cases they can be found to be inconsistent and changeable through time (van de Vall, 2015, Wharton, 2016).

This relates to broader challenges in how significance might be identified. A number of authors in the digital preservation community have raised concerns regarding the subjectivity implicit in the specification of significant properties by any one party (Dappert, & Farquhar, 2009, Yeo, 2010). Dappert and Farquhar suggest that significance is something assigned to a property by a particular agent or group and that this means that value judgements are implicit in their specification, while rarely discussed by those specifying them. Yeo suggests that significant properties defined by those caring for collections might not align with the needs of future stakeholders. This brings us to the question of whether we can make effective conservation decisions based on a concept as subjective as significance, particularly when limited to notions of identifiable ‘properties’.

5.2.2. Identifying Significance in Practice

In order to better understand how we might address problems with the use of

significant properties, as identified in the previous section, I will explore how notions of significance might find use in the selection of preservation strategies for software-based artworks. In this section I discuss three time-based media artworks—all of which employ software in their realisation—and consider for each how a weighting of significance might be applied when considering the kind of preservation approach to apply, particularly regarding how the artwork's identity might be separated from its software implementation. These three works were selected as they are all realised as projected moving images within an exhibition space. This allows an initial point of comparison from which to explore differences, which, as I will go on to demonstrate, arise in how the particular use of the software medium relates to the intentions of the artist, the creative process and the artwork's shifting context.

The first example is *The Clock* (2010) by Christian Marclay, which while not one of the core case study artworks chosen for this thesis—and not strictly speaking a software-based artwork—is helpful in illustrating one particular use of software. *The Clock* is a single channel video artwork that compiles scenes taken from cinematic history which portray time—a shot which is tied to a particular time through the presence of a watch or a clock face, for instance. The fragments of video are sequenced so that the appearances of time within the scenes flow in real-time, which can then be synced to the local time of the installation; thus rendering the work a functioning timepiece (White Cube, 2010). With respect to medium, *The Clock* very much operates within a cinematic framework, and as such its primary artistic medium is one of linear moving image in the tradition of artists' film and video art. Yet behind the scenes, software has been used to achieve the consistent playback of the considerable quantity of video data.

In this instance, software has no conceptual link to the artwork and no apparent presence within the artwork's realisation or any (viewer facing) descriptive information or documentation. While *The Clock* was realised using software in the vehicular sense, the software was not intended to articulate an artistic statement—which is instead located in the selection, editing and sequencing of the video fragments themselves (among other actions). The software is utilised here simply as a tool to achieve an effect. If this artwork were brought into a collection, this usage of software might remain relevant to the work's display in the short term, and in the longer term be historically interesting. Its absolute preservation however, is not important. Rather, preservation efforts would seek to ensure that the sequential playback of the video fragments could be maintained and synced to local time—the software

implementation used could be replaced with some other mechanism for achieving the same effect without impacting the work.

The second example I will consider is *Colors* (2005) by Cory Arcangel. *Colors* employs software which manipulates a video file, the processed output of which is projected into the exhibition space. The software program plays back each horizontal row of pixels in a specific video file (a QuickTime MOV of Dennis Hopper's 1988 film of the same name) line-by-line. Each line is stretched vertically to fill the projection area, which creates a shifting pattern of vertical bands of colour. Much like *The Clock*, this work's formal elements might be considered in relation to moving image mediums such as cinema (and in this case also to the history of artistic experimentation with video). When the work installed, there is little to immediately suggest from the projected image alone that software is involved in its realisation. However, this is complicated when we consider that Arcangel's practice has frequently engaged with coding (Arcangel, 2013, Arcangel, 2017), giving the presence of software a contextual and art historical significance. In contrast to Marclay's *The Clock*, this is a rather more ambiguous relationship between artwork and software, and we might consider intervention at the level of the software with caution when addressing the work's long-term preservation.

However, Arcangel takes a rather different attitude, explicitly stating that as far as he is concerned, the concept *is* the work and that he is happy for it to be reimplemented using different technology if necessary for purposes of long-term preservation (Arcangel, 2012, March 14). We might conclude from this statement that maintaining the actual technology of the original implementation—a Mac OSX application utilising the QuickTime and OpenGL frameworks—is not essential to the work's realisation in the future. In the case of *Colors*, the vehicle is the execution of code and the processing and manipulation of video data by a computer system, yielding output frames and audio. The artistic medium could be considered Arcangel's subversion of cinematic images and playful references to the slit-scan[18] technique. While there are considerations over the presentation of the work such as projection specifications and other display parameters, *Colors* would seem to be a clear example of a software-

[18] Slit-scan is a photographic technique which has been used for a variety of purposes, but the one referenced by Arcangel in *Colors* is its use to create abstract visual effects in cinematography, such as the 'Star Gate' sequence in Stanley Kubrick's *2001: A Space Odyssey*.

based artwork for which the authenticity of its realisation lies in the concept rather than the vehicle.

This conclusion is supported by an understanding of the artist's practice. Arcangel has talked about his artworks as DIY recipes (Birnbaum, & Arcangel, 2009) and has expressed an affinity with open source culture—much of his artwork source code is available online (Arcangel, 2013) and in printed publications (Arcangel, 2017)—where . Indeed, Arcangel has shared the source code for a version[19] of *Colors* online (Arcangel, 2017), exposing the mechanism and revealing the project's origins in a code template from the open source openFrameworks toolkit (anon. openFrameworks, 2018). Original and artist authored code is undoubtedly an important technical art historical artefact, however. In this case, in-line source code comments written by the artist—playfully identified with HTML-referencing <CORY> </CORY> tags—reveal how the code was extended from the original template. This also raises questions over how such technical art historical insight might be captured and reflected in public-facing documentation—a question I return to in Chapter 6. However, where achieving the ongoing realisation of *Colors* is concerned, it is what this code *does* rather than how it does it which holds primary significance. Faced with a choice between maintaining the integrity of the underlying implementation and being unable to display the work, migrating the code would likely be viewed favourably.

The third example is *Sow Farm near Libbey, Oklahoma 2009* by John Gerrard. This work depicts an agricultural complex on the American Great Plains, rendered in real-time 3D and simulating day and night cycles. The work was created with a 3D development tool and rendering engine called Quest3D (Act-3D, 2012), which might typically have found use in the creation of video games and architectural visualisations. The work was completed in 2009, and the results achieved represent high fidelity 3D rendering for the era it was created. While the work maintains a connection to moving image mediums through its use of similar visual language (for example, the scene is viewed from the perspective of a virtual camera), visual characteristics of the rendering techniques might invite connections with the vernacular of video games, for example. Gerrard is interested in what he calls a

[19] While this source code is actually for the *Colors Personal Edition,* a version of the work the artist distributed online, the code differs from the version collected by Tate by only a single line of code which only serves to skip processing of the black letter-boxing present in the source video file.

"'slippery' space between the real and the representation of the real" (Gerrard, 2015) and his use of 3D is integral to achieving this.

In the case of *Sow Farm*, the vehicle is the execution of encoded instructions and data and the resulting rendering of frames by the computer system. The artistic medium might be identified as Gerrard's articulation of his "slippery" representational qualities through the manipulation of the vehicle. This use of software is much more important than merely as a tool then—it is present in the artifice of the work's realisation and critical to its reception. We know from evidence of the work's production history that considerable effort went into achieving the precise characteristics of the rendered environment (Gerrard and Pötzelberger, 2015) and we might understand the identity of the work as residing in these carefully constructed details. The conservator might, therefore, be cautious in modifying the software involved and favour an approach which maintains the software as-is (e.g. emulation or virtualisation).

The previous assertion is complicated however, by the potential for the meaning of materials to shift through time. In comparison to the possibilities of 3D today, and indeed to Gerrard's own recent work, *Sow Farm's* 3D graphics are beginning to show signs of age. In Figure 15 below, I compare the 3D rendering of *Sow Farm* with that found in a work from 2017, *Western Flag*.

Figure 15. Comparison of 3D landscape rendering techniques in John Gerrard's *Sow Farm (near Libbey, Oklahoma) 2009* (left) and *Western Flag (Spindletop, Texas) 2017* (right). These images are detail from screen captures of the complete render in each case. © John Gerrard 2018.

While depicting different locations, so limiting one-for-one comparison, there are clear differences in the level of detail and realism achieved. In the more recent work textures are more detailed, lighting effects more sophisticated, and grass more realistically rendered. As the baseline measure of perceived realism in 3D graphics shifts, the "slippery" qualities of Gerrard's older works are at risk of being lost over time. This potential shift invites reconsideration of the potential significance of the artwork's original software implementation and poses the question: would it be desirable to attempt to augment the existing rendering pipeline (perhaps through post processing or porting to a modern 3D engine) in order to attempt to improve realism and maintain the link with viewer expectations?

When asked in an interview about technological change and the ageing of earlier works, the artist expressed his interest in the work's use of the technology of the time during which they were produced and felt that making any changes would need to be approached carefully (G. Gerrard, personal communication, 12 September 2016). Although he embraces technological change within his practice, he views the significance of the work as residing in the executable code he generated. This suggests that in this case, the successful display of this work in the future would appear to be contingent on maintaining the original software implementation as-is. In taking such as a decision, we would accept that a shift in meaning occurs in the decoupling of the material and the original context of its use. The identity of the work can now only be understood in relation to this history. How this kind of history might be recorded and conveyed to an audience becomes an additional concern for the conservator.

In the latter two examples examined above, it is clear that there can be considerable nuance to decision making in the conservation of software-based artworks. Understanding the significance of a particular use of software cannot be understood as relating simply to the intentions of the artist, nor indeed to any single narrative or account. Rather, significance is established through careful interpretation of sometimes conflicting sources of evidence—a process in which context plays a major role. Arcangel's artistic practice informs our understanding of the value of his code. Insight into Gerrard's shifting relationship with the representational qualities of his 3D environments helps us to navigate questions over the treatment of the software used. In the latter case, the identity of the work has shifted when seen in relation to how audiences would interpret the qualities of the 3D rendering. Thus we arrive at an understanding of significance and identity that is shaped not only by the properties of digital objects nor simply by the intentions of the artist, but by these factors viewed in conjunction with their evolving context.

5.2.3. Significant Knowledge

Published methodologies for capturing significant properties currently seem poorly suited to the scenarios outlined in the previous section, suffering either from being overly prescriptive at one extreme, or under-specified at the other. In digital preservation, various authors have suggested an approach which involves a constraint model of property-value pairs (Dappert, & Farquhar, 2009, Knight, 2009). In art conservation, Richard Rinehart's Media Art Notation System (MANS) takes the form of an XML schema which provides a score in order to "aid in the re-performance

or recreation of works of art" (Rinehart, 2004, p.3). While certain characteristics, such as the environment requirements for executing *Sow Farm,* might be usefully constrained to a set of values, the information required to appreciate the technical history of a work could not. Reducing the identity of the work to a digital object which can be performed risks loss of context and any historical record of the work being preserved.

At the opposite end of the spectrum to these highly structured approaches we have the Variable Media Questionnaire (VMQ) (Ippolito, et al., 2003). The VMQ provides an instrument for capturing artists' perspectives on the significance of their choices, and the identification of what its creator Jon Ippolito frames as "medium independent behaviours" (Ippolito, et al., 2003). Taking the idea of separating identity from technical implementation to a logical extreme, this is an open-ended framework for compiling documentation in collaboration with an artist. This flexibility is both benefit and drawback: benefit in that it permits considerable freedom in the formulation of resulting documentation; drawback in that no prompts for the capture of medium-specific information are provided, which creates a risk of missing important information. The principles of the VMQ and indeed, the artist interview in general, might have usefully gathered information relating to the case studies in the previous section, such as Gerrard's software-based artworks, but they still provide a relatively limited frame through which to understand identity.

Given that identity can often only be understood in relation to contextual information and tacit knowledge, and certainly not defined at any one moment in time, there seems to be a need for a broader framework. Potential components of this framework have already been proposed elsewhere. Guillaume Boutard and Catherine Guastavino propose the idea of "significant knowledge" as an extension of significant properties (Boutard, & Guastavino, 2012). Developed in the context of electro-acoustic instruments as cultural artefacts (which are similar to software-based artworks as technical systems with performative characteristics), this concept encompasses tacit knowledge and information about the creation of an artefact. The emphasis of their approach is on the intelligibility of the object of preservation, which sits in contrast to previous frameworks for significance which focus on rendering and authenticity.

Rather than disregard any of the other approaches discussed above—they all have potential value—I propose that we might completely reconceptualise significant properties as *significant knowledge* and so widen its scope. Through this simple

reframing of the problem, a great deal of the existing baggage is lifted from the concept. The emphasis shifts from attempts to distil identity into sets of properties or characteristics, to a more pragmatic approach of building knowledge that can support efforts to sustain the identity of an artwork through time. This is inherently less prescriptive than the approaches to significant properties we have available, and instead allows room for an interpretative and contextualised approach. For the purposes of this research, I define significant knowledge as:

> *The developing body of knowledge required to ensure the future realisation and intelligibility of a software-based artwork, in a way which can be accepted as authentic in relation to the original intellectual creation.*

While the form this knowledge takes is intentionally left very open, so permitting that it might to some extent reside in the tacit knowledge of individuals or organisations caring for the work, it is desirable for it to reside in concrete documentation materials wherever possible. Where knowledge can be made explicit in this way, there is a need for some kind of guiding structure, which I propose might be best served by categories of significant knowledge that guide this work rather than restrict it.

Two research projects have developed categories for significant properties for closely related domains, which might be easily extended to encompass significant knowledge. The first was developed by a conservation research team at Tate and proposes a classification for the significant properties of “networked art”, which might be considered a type of software-based art with strong network dependencies (Dipple, et al., 2010). This is a particularly useful source in its direct reference to the concerns of software-based art, and also in that it makes explicit the idea that the “identity of the artwork may be larger than the artwork itself” (Dipple, et al., 2010). The second comes from the software preservation domain, and a JISC-funded study of the significant properties of software (Matthews, et al., 2008). Also proposing a classification system, this study is important in its close consideration of the technical characteristics of software. However, the orientation of this study towards software as playback mechanism (i.e. a tool for rendering other files) rather than performed artefact, makes it less applicable to software-based artworks as-is.

Using a mapping of the Dipple et al. and Matthews et al. classification systems developed by Patricia Falcão, Time-based Media Conservator at Tate (Falcão, 2013), and further refined by myself, I have identified a set of seven categories of significant knowledge relating to software-based art. These categories are listed in Table 5

below with a brief description of their scope. I also provide a set of examples of the documentation materials which might support an understanding of each property category.

Significant Knowledge Category	**Tate Software-based Artworks Significant Property Categories (Dipple, et al., 2010) Mapping**	**InSPECT Software Significant Property Categories (Matthews, et al., 2008) Mapping**	**Significant Knowledge Description**	**Examples of Materials Making Knowledge Explicit**
Function	Behaviour Function Processes	Functionality	Knowledge concerning the intended functionality of the software (i.e. what it *does*) and how it manifests as a set of behaviours	• Artist's interviews and statements • Source materials and associated documentation • Development and design documentation
Experience	Rules of Engagement Visitor Experience	User Interaction	Knowledge concerning the experience of the work from the perspective of viewers or users (be that interaction in a physical setting or via a web browser, for example)	• Video documentation of previous realisation • Parameters for installation • Narrative accounts • Questionnaires
Structure	Content and Assets External Linkages and Dependencies	Software Composition Software Architecture Software Environment	Knowledge concerning the make up of the work including its constituent components (either physical or digital) and the relationships between them, and with their technical environment	• Source materials and associated documentation • Development and design documentation • Past installation documentation
Formal	Spatial or Environmental		Knowledge concerning the environment in which the work is intended to be	• Artist's interviews and statements • Past installation documentation

	Parameters Formal and Structural Elements		experienced (either physical or digital)	• User system requirements • Software analysis reports
Performance	Time Appearance	Operating Performance	Knowledge concerning the qualities of the software performance (such as timings or character of interactive elements)	• Software testing tools and metrics • Reference photographs, images and videos
Provenance	Other Versions of the Work Legal Frameworks	Provenance and Ownership	Knowledge concerning the lineage and versioning of the work and its components	• Version history • Ownership and rights Knowledge • Licence agreements
Context	Artist's Documentation of Process Context		Knowledge concerning the history of the work and its creation, and other contextual information that enhances understanding and intelligibility	• Source code and change tracking • Development and design documentation • Scholarly and critical writing • Press and media coverage • Social media data

Table 5. Identified significant information categories for software-based artworks, with mappings to related significant property frameworks and examples of supporting materials.

Of these categories there are four which relate closely to the software performance itself: Function, Experience, Structure and Performance. While the significant knowledge framework addresses these broadly, a more precise framework may be required to ensure that they are maintained when software or environment change in future realisations of the work. On the one hand we have works like *Sow Farm,* which demand maintenance of a tightly specified performance and so require detailed information about the technical environment in which this can be achieved. On the other, we have works like *Colors*, which theoretically permit a complete rewriting of the underlying software—so demanding a clear account of the precise functionality of software. Other works will sit somewhere between the two and so demand elements of both.

While the document examples listed in the table above serve to support significant knowledge relating to the identity of the artwork at the level of software performance, there would be considerable value in a unified, concrete approach to capturing relevant information as documentation. In the field of software engineering, this kind of information would be captured through the specification of requirements, which are formulated early in the design process and maintained alongside the software. In the second half of this chapter I explore how the principles of requirements engineering might clarify how significant knowledge regarding software performances can be made explicit.

5.3. Reframing Software Requirements

In software engineering, the sub-domain of *requirements engineering* is defined as the process of "finding out, analyzing, documenting and checking" the "services that a system should provide and the constraints on its operation" (Sommerville, 2015, p.83). In more general terms, these services might be considered the things that the software system *does*—corresponding to the idea of functionality introduced in Chapter 2—while the constraints are the parameters within which it must achieve those things. When producing a requirements specification document, these requirements are identified by or in collaboration with the relevant stakeholders in non-technical language (as far as possible), allowing the developers of the system to implement this functionality using their own technical solution.

The practice of requirements engineering emerged in the 1980s, partly in response to a crisis in software development as a result of increasing complexity, cost and scale of software projects around this time, and partly as an expanding range of users

became interested in the technologies involved (Alexander, 1997, Karch, 2011). Ian Alexander describes the shift:

> "Attention gradually moved, in software terms, from code to design, and then on to specification. This was understood initially as the precise description of components-to-be-built; gradually this understanding too broadened to encompass entire systems. Finally, with input from the human-centred sciences (psychology, sociology, ethnology...) specification has come to include a definition of the problem to be solved, as seen by the human users of any putative system." (Alexander, 1997)

The formalisation of eliciting requirements (what Alexander calls "the definition of the problem") from the *users* of the system directly—a problem-centric rather than technology-centric approach—was a particularly important innovation of requirements engineering. Requirements specification remains a ubiquitous component of mainstream software engineering today. Research into documentation methodologies among software engineers has revealed that requirements documentation is considered among the most important documentation artefacts for a software project in the context of ongoing maintenance (Lethbridge, et al., 2003, de Souza, et al., 2006). The core principles of requirements related processes are similar across different software engineering methodologies, even among those so-called agile approaches that eschew documentation in favour of speed and efficiency (Cao, & Ramesh, 2008, Inayat, et al., 2015)—the main difference being that these will be developed more iteratively.

Requirements are usually split into two types which I have already alluded to above: functional (the things the software should do) and non-functional (the constraints within which it should do them). The Software Engineering Body of Knowledge defines these terms as follows:

> "Functional requirements describe the functions that the software is to execute; for example, formatting some text or modulating a signal. They are sometimes known as capabilities or features."
>
> "Nonfunctional requirements are the ones that act to constrain the solution. Nonfunctional requirements are sometimes known as constraints or quality requirements."
>
> (from IEEE Computer Society, et al., 2014, p.1-3)

In a typical requirements engineering process, requirements would be defined in collaboration with stakeholders based on their needs, recorded in a document called a requirements specification and then the software solution developed would be validated against the these requirements (Sommerville, 2015). Requirements engineering principles also allow for management of change in requirements once a software system has been developed, as users demands on that system change (for example, a new feature is required).

I propose that requirements engineering principles, particularly the creation of a requirements specification document, can be effectively applied in support of the conservation of software-based artworks. The significance of requirements as a production artefact has already been noted by several authors working in the field of software-based art conservation (Engel, & Wharton, 2014, Marchese, 2011), but the concept has not yet been thoroughly explored as a process of documentation undertaken *after* production by the conservator. There are a number of reasons that the use of requirements specification and management may hold value in this context:

- **Separation of function from structure**: Requirements engineering aims to separate what is *required* of a system from any specific technical solution. In a software-based art context, this means that those elements of the software function or behaviour that are not specific to a particular technology can be identified and described in a technology agnostic way. This allows these elements of work to be reinterpreted or modified (within parameters specified by non-functional requirements) while maintaining the identity of the work, providing this has been understood as acceptable in relation to the artwork's material concerns.

- **Collaborative and non-technical:** Requirements are designed to be written collaboratively with stakeholders and in (as far as possible) non-technical plain English. This fits well with the collaborative nature of the conservation profession and with the demands of artist consultation and authorisation.

- **Management of change:** Requirements specification methodologies allow for the updating of requirements through time and the tracking of relationships between requirements and implementation (known as traceability). In the context of software-based art conservation, this might be necessitated when works are revisited, modified or migrated.

- **Communication with software developers:** Requirements specification is a ubiquitous documentation practice in software engineering and so will be readily understood by many software engineers and developers. This is an audience with which those caring for software-based artworks are increasingly likely to engage.

- **Synergies with conservation practice:** Eliciting and specifying requirements may have value beyond the creation of the requirements documentation artefact in itself, as the process may reveal further information about function, structure, experience and performance. It also acts as a historical record of changes through time.

The major difference between the typical software engineering process of requirements specification and the use scenario I am exploring, is that when software-based artworks are acquired the software has usually already been completely implemented. In this reframing of requirements principles, we are reverse engineering (a concept introduced in Chapter 4) the requirements from their implementation. This involves identifying, extracting and defining them using multiple sources of information including existing significant knowledge (such as documentation and the artist themselves) and analysis of the software and its environment. By specifying requirements in this way, we can identify the extent to which we can detach the concept of the software from its implementation. Problems with the application of requirements engineering approaches often involve poorly defined requirements (Firesmith, 2007, Cerpa, & Verner, 2009). In these cases, the implemented solution may not match the needs of stakeholders and achieve its intended purpose. These problems become much less relevant when requirements are generated *a posteriori*, as of course, the solution already exists—as such, it becomes possible to generate requirements that match the system. The only stakeholder negotiation required (while factoring in other considerations such as institutional resource constraints) is to understand which of characteristics of a software performance can be transformed back into less technology-specific requirements.

Requirements could theoretically be used to describe any aspect of a software system's behaviour, including the way in which it permits user interaction. In cases where this is particularly important it may be preferable to explore use cases and user stories as an alternative or complementary way of specifying requirements. These are similar approaches to requirements documentation which specify requirements in relation to a user's experience of interacting with that system. Use cases tend to

describe a user's interaction with a system through structured text and diagrams, while user stories are short semi-structured textual descriptions of a user's encounter with a system (Pressman, & Maxim, 2014, Sommerville, 2015). I suggest that an approach based on user stories may be most useful in a conservation context, as this methodology has emerged from agile development and is thus relatively lightweight and flexible. I discuss user stories in the context of an interactive artwork case study in Section 5.4.

In the following sections I discuss in detail functional and non-functional requirements respectively, and in particular investigate how they might be of value when used for describing software-based artworks of different types. I then illustrate two particular practical applications of requirements specification through two case studies which relate primarily to functional and non-functional requirements respectively.

5.3.1. Functional Requirements

In this section, I look at how we might understand what the software components of software-based artworks do—their functionality—and how this might be captured as functional requirements. Introduced in the previous section, functional requirements are those requirements that specify the functions that the software is meant to execute. One of the primary questions in applying this principle to software-based artworks is: to what level of detail should this functionality be specified?

Most simply, functionality could be described with a statement of the purpose of the software component of a software-based artwork. In Table 6 below I compare such statements for the six artwork case studies. These statements are based both on existing documentation of the works held by Tate (including artists' contributions) and my own experiences of examining and analysing the software involved. It should be noted that they refer only to the software component of the artworks, which while always of primary importance, may form only part of a more complex system or assemblage.

Artwork Title	**Artist**	**Description of Functional Purpose**
Becoming	Michael Craig-Martin	The function of the *Becoming* software is to render a dynamic arrangement of 2D objects to a display device of fixed size. It must ensure the objects' correct relational arrangement and randomise the fading in and out of the objects, including the length of time taken to fade in and

		out.
Subtitled Public	Rafael Lozano-Hemmer	The function of the *Subtitled Public* software is threefold: 1) it must locate and track visitors to the exhibition space using a video feed from CCTV cameras; 2) it must project randomly selected words from a predefined list onto the tracked visitors and allow the exchange of assigned words when two individuals come into close proximity; 3) every few minutes the projection must briefly switch to the raw video camera footage.
LiMac Museum Shop [website]	Sandra Gamarra	The function of the *LiMac* software is to manage, store and serve an internet accessible set of web pages (including scripts, styling and image media) managed through a content management system.
Brutalism: Stereo Reality Environment 3	Jose Carlos Martinat Mendoza	The function of the *Brutalism* software is to search the internet for the word 'brutalism' (sometimes with an additional accompanying search term), harvest results and convert them into simple paragraphs of text, and then print these results onto small slips of paper.
Colors	Cory Arcangel	The function of the *Colors* software is to play back each horizontal line of pixels in a video file frame by frame (with the sound played back as normal), stretching the pixels vertically to fill the screen. After playing each line of pixels in the video file, the software should repeat this process.
Sow Farm (near Libbey, Oklahoma) 2009	John Gerrard	The function of the *Sow Farm* software is to realistically simulate a pig farm and surrounding environment in real-time, using a 3D visualisation engine and according to the precise formulation of the artist's expression. The rendered environment will be presented from a slowly orbiting camera. The simulation should run indefinitely, and incorporate the animation of the arrival and departure of a truck which is triggered once every 159 days.

Table 6. List of software-based artwork case studies and simple descriptions of the functional purpose of their software component.

The strength of these short functional descriptions is a clear articulation of purpose of the software, but is this sufficient to allow the reinterpretation of the work if future

conservation treatment demands it? Looking at what is perhaps the most computationally straightforward work on the list, Cory Arcangel's *Colors*, we might think it is. Arcangel has quite explicitly stated that *Colors* can be considered an "algorithm" of sorts (Arcangel, 2012, March 14), and that there is no expectation that the desired effect be achieved using any particular technology in the future. We might want to know slightly more detail—which could be relatively easily determined through the analysis of the small code base—such as where the pixel scanning begins and whether the output frame rate should match the video file, but otherwise it is easy to conceive of a reimplementation that achieves identical results to the original.

Looking at more complex software, such as that supporting *Subtitled Public* (consisting of many thousands of lines of Delphi code), we find it more challenging to capture the work through a simple statement. Referencing the functional description presented above with an actual installation of the work would raise a number of questions. In what font, colour and size should the words be projected? Should the accuracy and quality of the tracking and projection reflect technology at the time of the works creation, or be updated to improve performance? Should the word list be updated or added to depending on the context of the installation? To support the answer of questions such as these, there is a need to develop a more sophisticated model of functionality documentation, particularly in relation to the nuances of behaviour which are not made explicit in the existing documentation.

This is where the capture of more granular statements of functional requirement may be effectively applied. There is some flexibility in how these requirements are actually captured, but each statement of functional requirement should include as little ambiguity as possible. Although in practice there is no single accepted template for requirements specification, the process has been made an international standard (within the ISO framework) by the Institute of Electrical and Electronics Engineers and International Electrotechnical Commission (anon. ISO/IEC/IEEE 29148:2011: Systems and software engineering — Life cycle processes — Requirements engineering, 2011). The most recent version of this standard defines three templates which aim at capturing slightly different levels of detail. Of these I propose that the lowest level approach—the Software Requirements Specification (SRS)—may be the most appropriate in order to maximise the information captured. The standard defines this as "a specification for a particular software product, program, or set of programs that performs certain functions in a specific environment" (p.45). The guidelines for producing an SRS do not specify that functional requirements should take any specific

form, but states that they should describe “the fundamental actions that have to take place in the software in accepting and processing the inputs and in processing and generating the outputs” (p.58). The value of requirements would be enhanced by ease of use, and this flexibility removes the barrier of a formal syntax and may help ensure their capture regardless of any one conservator’s approach. In Section 5.4 I return to the *Subtitled Public* case study introduced above to explore its functional requirements in more detail.

There are cases (even complex ones) where functional requirements may not be a useful way to document a software-based artwork, or at least provide limited value. This is likely to be most apparent where works are “thickly” specified (see Section 2.3 for an introduction to this terminology) with regards the specific software technology employed. John Gerrard’s *Sow Farm*, which is a particularly clear example of this kind of work and serves to illustrate this point. This work was realised in a 3D engine representative of the technology of the time it was produced, called Quest3D (Act-3D, 2012). As a result, it presents visual characteristics in the rendered 3D environment, which require that it is realised in this specific engine in order for them to be maintained, and thus maintain this aspect of its identity. This severely limits how much value there would be in specifying the complexity of the engine as functional requirements (for example, the way in which grass is rendered using an adapted fur shader), as they may be very difficult to describe accurately or recreate in contemporary 3D engines. In this case it is more appropriate to maintain the software exactly as it is (so including its visual characteristics), while maintaining an appropriate technical environment in which it can be performed (for example, by emulating this environment on contemporary hardware).

As a result, *Sow Farm* could be specified with a single functional requirement which makes direct reference to the technology used: *the software must simulate and render the Sow Farm 3D environment from the associated data assets in the Quest3D engine according to the associated data structures contained in the files acquired from the artist*. This is, of course, a somewhat redundant act of documentation—it offers little value beyond that which can gained from even a cursory examination of existing documentation. Migration or reinterpretation would not be an appropriate preservation strategy for this work and therefore we are likely to look to techniques such as emulation and virtualisation to achieve long-term preservation. When applying this kind of strategy, requirements relating to performance and rendering quality become much more significant concerns in achieving an authentic realisation

of the work. For this work, and other similarly thickly specified software-based artworks, identifying and capturing these non-functional requirements should be prioritised.

5.3.2. Non-functional Requirements

In this section I will explore the constraints on quality or performance that might be linked with functional requirements, and how these might be captured as non-functional requirements. While functional requirements are the things the software does, non-functional requirements specify the way in which it should do those things. Unlike functional requirements, non-functional requirements might also be associated with metrics and operate within ranges or bounds of acceptability. There are a large number of kinds of non-functional requirement, and while no single standardised classification exists, this topic has been well explored in the software engineering literature (Chung, et al., 2000, Glinz, 2007, Chung, & do Prado Leite, 2009). In the context of documenting software-based art, I have identified the following kinds of requirements as of primary concern, presented below with examples:

- **Performance** (e.g. a consistent level of response time to interaction must be maintained; frames must be rendered at a rate of at least 30 frames-per-second);
- **Quality** (e.g. certain post-processing effects must be applied; vector graphics must have a certain kind of anti-aliasing applied);
- **Reliability and Stability** (e.g. the software be able to run for a certain length of time independently and without fault; the system must be able to suffer power failures);
- **Security** (e.g. if the software is connected to or presented over the internet it must be appropriately secured; if interfaces are accessible to gallery visitors they must be securable to prevent tampering).

Addressing the capture of these kinds of requirements necessitates a thorough understanding of functionality, and in many cases, the structural components of the software and the parameters of its previous realisations. The ISO/IEC/IEEE standard for requirements engineering, as for functional requirements, does not specify any particular format for their capture, but does emphasise the identification of “the verification approaches and methods planned to qualify the software” (anon.

ISO/IEC/IEEE 29148:2011: Systems and software engineering — Life cycle processes — Requirements engineering, 2011, p.61), a topic I will return to below.

In contrast to functional requirements, when identifying non-functional requirements it is particularly important to work outside of modes of experiential essentialism (a concept introduced in Chapter 2) and to address the underlying software processes. Artworks which rely on graphics rendering are an example of a kind of software experience which focuses on the screen (or projection), and so obscures the complex software processes that create this manifestation. Non-functional requirements relating to rendering are particularly relevant for software-based art due to the prevalence of artworks producing visual output or carrying out image capture and processing. This rendering pipeline is a consideration in the realisation of four of the seven case study artworks examined in this research (*Becoming*, *Colors*, *Subtitled Public* and *Sow Farm*).

The transformation of code and data into image frames, then rendered and delivered through an output device, depends on a graphical rendering pipeline that is made up of many interlinked software and hardware components. These include physical graphics hardware and associated drivers, operating system supported interfaces to allow communication between software and the OS kernel, drivers and specialised hardware components. The relationships between these may need to be carefully disentangled to capture their performance and quality requirements, and appropriate tools identified for their later verification. In Section 5.5 I use the artwork *Sow Farm* as a case study to demonstrate how these challenges might manifest, while related software analysis methods are discussed further in Chapter 4. Even for works such as *Brutalism*, which involves no screen or projection outputs in its realisation, issues of rendering can still be relevant. In this case, the Ubuntu configuration employed uses the Gnome GUI. Understanding that this requires access to a display driver in order to be loaded was essential in creating a virtualised version of the software and its technical environment.

The *Sow Farm* case study represents a work for which machine-driven verification of non-functional requirements could be usefully applied to address rendering performance and quality concerns. However, there are cases in which this kind of approach may be less useful. *Becoming* is a relatively computationally straightforward piece of software. It is not interactive in any sense after the software has started running, and runs continually after this point (unless interrupted) in a single state—which is to say, it can be considered either on or off. This simplicity of function places

an emphasis on the rendered result and adds particular weight to issues around performance and quality requirements. At the concept layer we might see the objects rendered in *Becoming* as line drawings, conceptually and stylistically similar to those the artist uses in his wall-based works (which are realised in various media, but typically drawn or painted). At the logical level however, these are understood as 2D scalable vector graphics. The vector graphics are handled by code written in Lingo and are embedded in a Windows Portable Executable file containing Shockwave projector and the requisite dependencies.

The 2D assets have been acquired alongside the work as supplementary materials, and so can be examined. A cursory glance at these 2D graphics on a contemporaneous system would likely indicate that the files are identical to those embedded in the executable, and that the object rendering could be documented as the functional requirement: *the software must be capable of rendering the associated SVG vector graphics files.* However, an understanding of the SVG format reveals that their rendering can be subject to renderer specific edge anti-aliasing, resulting in distinct visual characteristics to the edges of the shapes (anon. Web technology for developers - SVG attributes: shape-rendering, 2014). This could be documented as a non-functional requirement which specifies: *the software must anti-alias the edges of the SVG vector graphics to conform to the anti-aliasing algorithm applied in the original (2003) realisation of the work.* This kind of non-functional requirement might be difficult to verify by addressing the software at a technical level—there is no means of programatically measuring SVG anti-aliasing in a Shockwave projector file. Instead, it represents a case in which visual documentation, such as a lossless video screen capture of the work, might better serve this goal.

It is helpful to consider execution environment and abstract dependencies in relation to technical requirements, which could be modelled as part of the requirements specification (see Chapter 4). Technical requirements are a specification of the individual components required in order to successfully perform a software program. For commercial software, these are often provided as abstract requirements specifying an acceptable minimum or range of power or performance - for example, a program might require 8GB of RAM or more. In reality, it may be hard to derive these requirements. The artist or gallery supplied machine is sometimes quite precisely specified by the artist, but might also just be a suitably specified machine available at the time of fabrication or sale. Furthermore there is unlikely to have been much testing on other systems to yield comprehensive technical requirements. In

these cases, the specifications or the artist approved version may provide a safe minimum and further alteration be made cautiously.

5.4. Case Study: Specifying an Interactive Artwork as Requirements

The artwork *Subtitled Public* by Rafael Lozano-Hemmer was introduced earlier in this chapter, and is highlighted here as complex software-based artwork with an identity which resides primarily in its functionality—so making it suitable for documentation using functional requirements. To briefly reiterate the earlier stated functional description: the work is an interactive installation, which projects a single random word (from a predefined list of conjugated verbs) onto each visitor to the exhibition space it is installed in. This word follows this visitor around the exhibition space, and can be exchanged with another visitor's word when the two come within a certain distance of each other. In a user manual created by the artist, there are some notes on the works preservation which include the statement:

> "From the artist's perspective, the project as it is now[20] is beautiful and delivers the required effect. However, the artwork is not the tracking system and algorithms currently used but the concept of subtitling the public. In this sense he is open to future ways to accomplish the effect." (Lozano-Hemmer, 2006, p.24)

It would seem a high priority then, that the conservator handling the works care understand what exactly the "concept of the subtitling the public" is in clear terms. I propose that this could be captured using requirements specification. *Subtitled Public* is a very well documented piece, but as I will go on to demonstrate, limitations to the original documentation are discovered during the process of specifying formal requirements. There is some context required to ensure that this following analysis make sense, including the definition of some essential terminology. In practice, such a terminological clarification might be presented at the beginning of a requirements document (anon. ISO/IEC/IEEE 29148:2011: Systems and software engineering — Life cycle processes — Requirements engineering, 2011).

The work is assumed to be presented in what I will term an *exhibition space* (which is also the artist's original phrasing), taken to be a relatively large (at least 9 x 9 x 4 meters), darkened, open room. The members of the public that enter the space to

[20] We presume the artist is referring to the 2005 version of the work, as it was acquired by Tate.

experience the work will be referred to as *visitors*, and the words that are projected onto visitors as *subtitles*. The exhibition space is divided into *zones*, each of which contains a set of linked components called a *surveillance pod,* consisting of a camera, computer and projector. These need not be maintained as discrete units (the camera for example, if often in the middle of zone while the projector is on the edge) and in fact, due to the low ambient light, equipment is not actually visible in the installation other than as in relation to the light emitted by projectors. For the purposes of this analysis, detailed non-software requirements (e.g. ceiling height, carpeting, wall painting) are assumed to have been captured in separate installation documentation. The work has two modes, *tracking mode*, which is when subtitles are being projected, and *video mode* when the raw video feeds are being projected.

The requirements identified below are based on extensive documentation provided by the artist and generated by Tate, as well as on an examination of the software executables, their source code and mock installations of the work. The functional requirements for the software components of *Subtitled Public* (i.e. what it is required to do) could be specified as follows:

- Individual visitors arrival and movement within the exhibition space must be tracked.

- Subtitles must be projected onto individual visitors from their arrival, and the position of the subtitles in the middle of their chests maintained as they move about the exhibition space.

- Subtitles must be selected at random from a predefined list of words (conjugated verbs) and the same word should not be projected more than once at the same time.

- When two individual visitors come within a user definable distance of each other, their respective subtitles must be swapped.

- Video cameras must be used to capture live video of visitors to the exhibition space.

- Every three minutes the projectors must project the raw camera feeds into the exhibition space for a user determinable amount of time, and then resume subtitle projection where it was left off before the switch.

- An administrative user must be able to modify the set of words and add new

words.

- An administrative user must be able to switch between word sets, which represent different languages.
- It must be possible for an administrative user to control the software system from an accessible location while the work is being exhibited.

The non-functional requirements for the software components of *Subtitled Public* (i.e. the constraints on the functional requirements identified above) could be specified as follows:

- The subtitle should use the following font specification:
    - Font: Arial
    - Font style: Regular
    - Size: 8
    - Script: Western
    - Colour: #C0DCC0 (hex) or R192, G:220, B:192 (RGB) or H:120, S:13, V:86 (HSV)
- The highest projection resolution possible should be used to ensure that subtitle fonts are smoothly anti-aliased.
- Subtitle text should be appropriately scaled to ensure that they are contained roughly within the body of a visitor, and therefore remain readable by other visitors.
- Subtitles should be projected at chest height (from the floor or feet of the visitor).
- The software should run stably and without interference required once initialised, for as long as the exhibition space is open.
- The subtitle projection should refresh at a rate which results in smooth tracking that keeps pace with an individual’s movements within the gallery.
- The software should be able to simultaneously track as high a number of visitors as possible.

We can also specify requirements as user stories[21], a notion I introduced in Section 5.3. For an artwork such as *Subtitled Public* which involves interaction at its core, this may be particularly valuable in understanding the nature of this interaction and identifying problems which may arise in maintaining its characteristics. This short example imagines a hypothetical gallery visitor's experience as a sequence of events:

- When a visitor enters the exhibition space they should be immediately identified as a new object to track, a random word fetched from the predefined list and (if the system is in tracking mode) a subtitle projected onto them at chest height.
- As the visitor moves freely through the gallery space this subtitle should follow them and be positionally maintained at chest height.
- If the visitor touches another visitor, this should be identified immediately, their assigned words exchanged and the projection updated.

In this case, the specification of a user story raises considerations missed in earlier requirements specification. The focus on interaction reveals that we must consider the response time of the system when a visitor enters the exhibition space and when two visitors come into proximity and exchange words. While it is otherwise somewhat limited in what is captures, the user story is in this case complementary to more fully fleshed out requirements.

To illustrate how requirements specification can help separate the core identity of the work from its past realisations, we can look to what is *not* covered in the requirements statements, particularly in contrast to aforementioned technical documentation such as the user manual. This, by inference, is detail which is not essential to the future realisation of the work. The model of camera, the specific computer hardware and the actual software implementation itself are not important to the realisation of the artwork. They may be of historical and technical interest, and therefore preserved, but they need not be maintained in their current form when realising the work in the future or where changes are required to keep the work realisable. It would also be theoretically permissible to improve the software performance's alignment with the desired non-functional requirements. For example, existing problems with the

[21] While these might typically be written from the perspective of a user, in the example that follows I have written from the perspective of a system designer.

tracking software, such as its inability to reliably identify chest height based on the height of a visitor, could be addressed.

Specifics of the exact tracking mechanism are also conspicuously absent from the requirements, but investigating this issue reveals that software requirements alone should not be relied on—or at least, that they should allow for a degree of interpretation. In its 2008 realisation at Tate Liverpool, the piece used infrared-sensitive cameras to improve tracking, the performance of which is boosted by the use of 'congo blue' filters applied to the rooms lighting. The artist has specified that the parameters of this lighting are in theory flexible, including the colour, provided the artist is consulted. We can therefore infer that the software using infrared is not a requirement of the work either, allowing the potential for other tracking systems to be employed. The artist has expressed an interest in the Microsoft Kinect2 capture device to these ends, suggesting that its "tracking is orders of magnitude faster, more accurate and easier to install" (Lozano-Hemmer, 2015). It also includes the requisite video feeds.

While the artist has clearly stated his interests, given the ageing software and challenges of installation, we might question whether such a change might also result in the loss of some of the identity of the work as represented by the 2005 version. The speed of tracking observed, the qualities of the blue-hued low light and the character of the raw camera feeds all add up to a very particular experience which is closely linked to the nature of surveillance technology at the time the work was created. These characteristics, one might argue, are core to the identity of the work. If requirements are unable to capture this kind of nuance, then can they be relied on? While this demonstrates the risks of considering requirements in isolation, an appropriate solution would be to specify more granular requirements. The tracking speed could be constrained as a non-functional requirement, while the cameras could be specified to only be models within a certain range of performance and image quality—and these requirements could be associated with video footage of past installations. In this case, establishing connections between requirements and materials capturing significant knowledge enhances the value of the former.

5.5. Case Study: Consistent Rendering and the Verification of Non-functional Requirements

As I argued in Section 5.3.1, *Sow Farm* is not a work which is usefully represented by functional requirements. As discussed elsewhere in this thesis, this work is likely

to be best preserved in its current software implementation in the Quest3D engine, in order to maintain the specific graphical qualities of the work. This shifts the emphasis of requirements analysis to the non-functional requirements that constrain the performance of this work. This case study demonstrates the process of capturing and verifying such requirements for a complex work reliant on the rendering pipeline. *Sow Farm* is a work which, even seen in the light of technological advances since its creation, employs sophisticated 3D rendering techniques, which have been very carefully applied by the artist and his production team. Maintaining these is, as discussed in Section 5.2.2, essential to maintaining the identity of the work. Much of this character is located within the binaries and associated data, but the technical environment in which execution occurs also plays an important role. With this in mind, there are two non-functional requirements relating to the projected output of the software that I will consider in this section: rendering speed (measured in frames per second) and graphics settings applied at the driver level.

The rendering speed requirement might be specified as: output must render consistently at a consistent 60 frames per second. One of the primary measures of performance for 3D applications such as a *Sow Farm* is the number of frames rendered per second (FPS). This metric has its origins in moving image and is used in characterising film and video, where set rates (e.g. 24 FPS for 35mm film) exist for particular media formats. A digital video file for example, will have a certain number of frames stored in an encoding format and a player will attempt to play them back at the speed determined by this format. As the system resource requirements of this process (understood in relation to the capabilities of the CPU and graphics card) are relatively light in the case of video, this is usually easy to maintain (although modern high definition formats may challenge this requirement). A real-time 3D application on the other hand, while also experienced as frames which are rendered and sent to an output device, does not have a predetermined number of frames. Frames are generated on-the-fly by the graphics processing hardware, based on instructions from software. Achieving a high and consistent frame rate is usually considered the most desirable level of performance for real-time 3D applications, and this is also the case for *Sow Farm*.

Sow Farm has a number of dependencies which may result in it no longer functioning on contemporary hardware in the near future (these are explored further in Chapter 4). In order to plan for the future and keeping the artwork running in new computational environments, it was proposed that the work be virtualised, and initial

experiments were carried out by a research team at Tate in 2015 (Falcão and Dekker, 2015). This approach would be advantageous for preservation due to the potential for generalising the software's dependencies—for example a virtual graphics card could be used instead of specific hardware. While the virtualisation of 3D applications is still in its formative stages, some consumer level virtualisation platforms such as VMware Workstation (VMware, 2018) support graphics processing through a virtual SVGA display driver, which mimics the functionality of a graphics card and its driver. However, this uses emulated video ram and so is likely to exhibit lower performance levels than a real graphics card, which is larger and designed to efficiently calculate math operations common in 3D rendering. Despite this potential limiting factor, it does allow use of the DirectX 9 framework required by *Sow Farm*. In fact, when installed in a virtualized Windows 7 environment, the application was found to run at what seemed to be a high, consistent framerate according to frame rate measurement tools.

However, there was nonetheless a visible impact on performance, perceptible to the human eye as an occasional subtle drag of the motion of the camera. This did not seem to be reflected in either of the FPS monitors logging outputs, which recorded FPS at a fairly consistent 125-130 FPS. These tools included one built into the Quest3D software itself and an independent monitoring program called RivaTuner Statistics Server (Hagedoorn, 2017). It was only through an examination of *frame time* values, a less frequently used performance metric which measures the length of time taken to render each frame (in milliseconds), that the limitations of the FPS metric were realised. Logging frame time, it was revealed that on this more granular level, some frames were taking double or triple the amount of time to be generated when the software ran in a virtual machine, in contrast to a consistent frame time for the native installation. Values from logs recorded for the native and virtualised version are plotted in Figure 16 below for contrast.

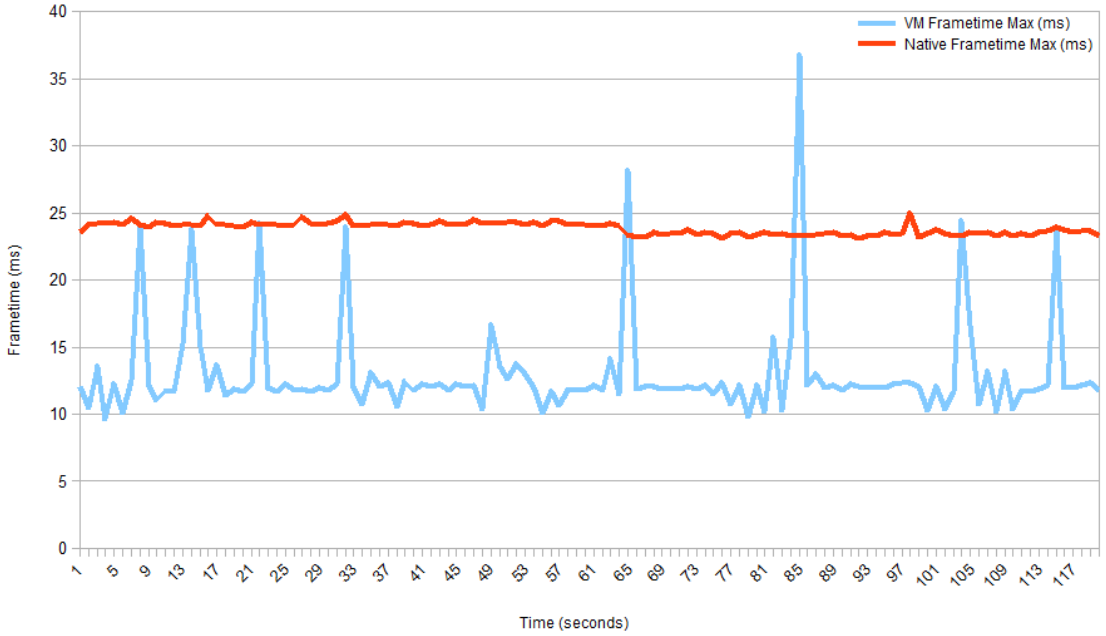


Figure 16. Line graph plotting frame time values (ms) against running time for the *Sow Farm* software running in a VMware virtual machine (blue) and natively on the host machine (red). Logging of frame time values was carried out separately for native and virtual environments.

The native version consistently generates frames at a rate between 23 and 25 milliseconds, while the virtualized version occasionally shows dramatic spikes in frame time. These spikes were sufficient to cause a perceptible drag in the motion of the camera in the screen output. In this case, problems in achieving a software performance were identified by a human viewer and clarified through closer examination of the technical properties of the software. As a result of these processes, a potential conservation treatment was rejected. The non-functional requirement could now be phrased slightly differently, and state: the output must render at a consistent 60 frames per second, *and with a variance in frame time of no more than 2 milliseconds.*

The second non-functional requirement I will consider relates to the quality of the rendered image, in the use of the driver level graphical configuration options. The requirement might be expressed as: *specific NVIDIA display settings must be applied to the rendered 3D image at the driver level (4x multi-sample anti-aliasing and 16x anisotropic filtering).* In order to achieve this, a crucial element in creating an appropriate technical environment for the software performance is the configuration of custom display driver settings for the graphics card hardware (also known as a graphics processing unit or GPU). An appropriate GPU chipset model and associated driver made by a particular manufacturer (in this case NVIDIA) can be used to force-

apply these graphical effects for a specific software application, though they are generated by the driver not the application itself and are therefore contingent on this configuration being applied when the software is placed in a new technical environment.

In this case, these settings result in noticeable changes to the graphical rendering of the 3D environment, as illustrated in Figure 17, which features two screen captures of *Sow Farm* running on the same Windows 10 desktop computer with and without the two NVIDIA driver-level settings applied.

Figure 17. Comparison of frames from two performances of Sow Farm, one with default NVDIA display driver settings applied (top) and the second with custom NVIDIA display driver settings applied to force multi-sample anti-aliasing and anisotropic texture filtering (bottom).

Multi-sample anti-aliasing smooths the jagged edges of 3D objects and in this case has a particularly visible impact on the right-most telephone pole. Anisotropic texture filtering improves texture quality on surfaces viewed from oblique angles, and in this case has a particularly significant impact on the detail present in the grass in the foreground.

The maintenance of these display settings should be considered an essential part of the correct performance of the software. However, in most virtualisation environments these particular effects are unsupported by existing virtual display drivers. Furthermore, the settings utilised may be specific to the driver version used (or a range of versions). It is also quite possible that future versions of the NVIDIA display driver will drop support for older features in favour of new methods, and so compromise the aesthetic provided by the older settings. Given that virtualisation and the use of a generic VGA driver is not yet an option, this raises questions over whether these should be applied if they become available without impacting other aspects of the work's identity. Would a VMware implementation of anti-aliasing match the qualities of the one available through the NVIDIA driver? This further emphasises the importance of negotiating the fine detail of non-functional requirements with the artist, even where the level of functional change between realisations it expected to be low.

5.6. Chapter Summary

In this chapter I have advanced a theoretical framework for capturing the identity of software-based artworks. Revisiting the significant properties concept from the field of digital preservation and establishing links with related ideas from art conservation, I found that existing approaches suffer from various problems which make their use in practice difficult when applied to software-based artworks. In many cases they are overly prescriptive and conditional on the reduction of a work into a set of properties which fail to capture the rich context within which the artwork continues to evolve. I propose the notion of significant knowledge as an alternative, which shifts the emphasis from properties as constraints to knowledge (be it tacit or explicit) that supports the understanding of the artworks evolving identity. Combining two existing classifications of significant properties from related domains, I propose a set of categories of significant knowledge which might serve to guide efforts to ensure it is representatively captured.

With this theoretical foundation in place, significant knowledge relating to the software performance was identified as requiring further consideration. For this purpose I

proposed a reframing of the requirements engineering process, a ubiquitous component of software engineering practice which describes the problem the software should seek to solve. Taking an approach which aims to reverse engineer requirements from implemented software, I found that its principles can be used to usefully articulate various issues relating to software-based artwork identity. The functionality of the software can be specified in a technology-agnostic way using non-functional requirements. Non-functional requirements can be used to effectively describe constraints on the parameters of a software performance. While requirements templates from software engineering may not be suitable for use in a conservation environment as-is, the principles of requirements engineering alone may offer a valuable conceptual core for the documentation of software-based artworks. The extent to which a software program can be transformed into requirements appears to be variable, and they must remain supported by contextual materials and other relevant components of significant knowledge.

CHAPTER 6

DOCUMENTING THE EVOLUTION OF SOFTWARE-BASED ARTWORKS

6.1. Chapter Outline

In the previous two chapters I have focused on developing approaches to documentation that capture some aspect of the software-based artwork at a particular moment in time—that is, they provide a kind of snapshot. In Chapter 4 this was the analysis of a realisation of a work, in order to generate a representation of the software structure employed. In Chapter 5 this was the use of documentation to capture the identity of a work and the software performance itself. An underlying assumption of these discussions has been that software-based artworks change through time, yet how this might actually manifest has not yet been explored. In this chapter I focus in on the processes of change that a software-based artwork might experience and consider how its ongoing evolution might be captured as documentation. This kind of documentation has the potential to support assertions of authenticity and capture the technical art history of a work for future study.

I will start by assessing how we might conceptualise the life of a software-based artwork, by examining existing models that can be characterised as lifecycle and

continuum approaches. In order to problematise these characterisations, I look closely at the set of processes involved in creating and maintaining software-based artworks and software performances. I particularly consider the contrasting nature of low-level incremental processes of change, which typically occur at the level of code, and the higher-level transformations that yield discrete versions. Considering existing documentation models from computer science and information science, I consider the extent to which these processes can be transformed into useful documentation. Finally, I explore a perspective on change documentation which unifies continuum principles with the notion of biographical accounts of artworks, and which may provide a means of capturing the software-based artworks movement through complex socio-technical dimensions during its life both inside and outside the collection.

6.2. Conceptualising the Lives of Software-based Artworks

In trying to identify how change manifests for software-based artworks, it is helpful to characterise what the life (used here to refer to the length of time an artwork, or a trace of it, exists in any tangible way) of such an artwork might be like in terms of the creative processes that shape it and the changes that occur from the point of its first realisation. From there, we can begin to identify the kinds of process which result in change, the levels at which they occur and when they should be documented in the course of caring for the work. In this section I consider two conceptualisations of the life of a digital preservation object, which offer contrasting philosophical perspectives on the relationship between this object and the processes that shape its existence. These are *lifecycle models* and *continuum models*. Below I introduce each perspective, and consider their potential benefits and limitations in understanding and capturing the lives of software-based artworks.

A lifecycle model can be broadly characterised as implying discrete phases through which the entity in question passes during its life. Luciana Duranti has pointed out that while lifecycle models are often construed as relating to ideas of human life, they are in fact employing the metaphor of circular natural resource cycles (such as the carbon cycle) (L. Duranti in Ashley, et al., 2015). Despite an identifiable shared basis, lifecycle models can take very different forms. In order to draw out some of the key characteristics of lifecycle approaches I look at three models of this type, developed in the fields of digital preservation, art conservation, software engineering respectively.

The Digital Curation Centre’s (DCC) Lifecycle Model emerged from the digital

preservation domain and the work of the eponymous institution, which claims the value of a lifecycle approach is in ensuring "that all the required stages are identified and planned, and necessary actions implemented, in the correct sequence" for curation and preservation of digital material (Higgins, 2008, p.135). While many components of the model seem aimed as modelling research data and simpler digital objects, it theoretically permits understanding of "complex digital objects", as we might consider software-based art to be. This model is represented as a series of layered, concentric circles which are cycled through clockwise (illustrated in Figure 18 below).

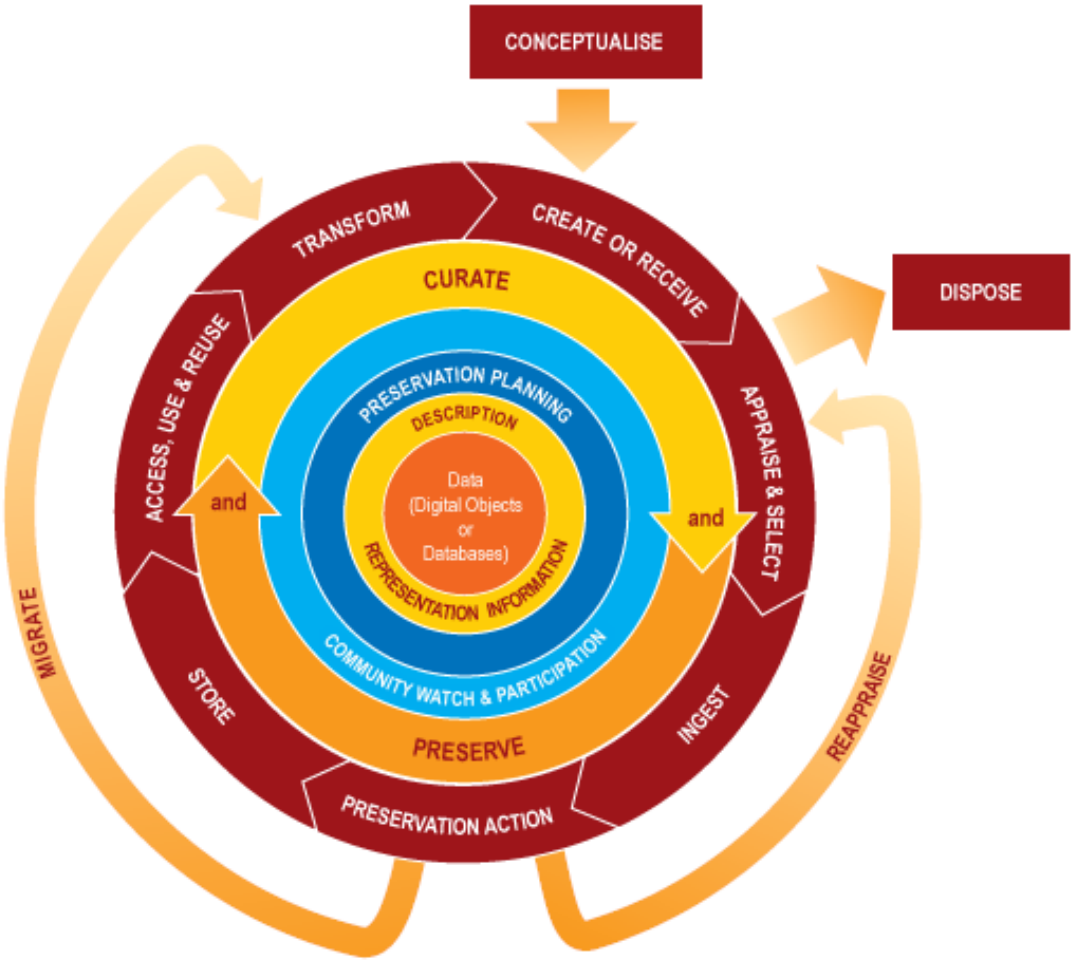


Figure 18. Representation of the Digital Curation Centre's (DCC) Lifecycle Model, reproduced from Higgins, 2008.

All processes within the DCC model occur after the creation of the digital material, and it is implied in its specification that the state in which the material enters the curation and preservation environment is to some degree fixed in terms of its identity (although on a technical level, it might later be transformed using migration or other processes). This would present an immediate problem for considering software-

based artworks within such a lifecycle. As described in Chapter 5, the software-based artwork is understood in relation to an identity that is heavily context dependent and might shift through time. While the sequential ordering of the outer ring phases is presented diagramatically as linear, the model does incorporate non-linear elements, such as the “migrate” and “reappraise” pathways, and parallel occurrence of phases implied by the concentric “preservation planning” and “community watch and participation” rings.

The DOCAM documentation model (introduced in Section 3.3.1.1) from the time-based media art conservation domain also specifies a lifecycle component, but in this instance the authors acknowledge that “media artworks tend to follow dynamic and vastly different lifecycles”, and so specifies a slightly less linear model (DOCAM, n.d.). This model centres on a “work” (i.e. artwork) instead of a digital object but still incorporates many of the same broad process types as the DCC model. The key difference however, is that it does not specify a sequential ordering. Instead, lifecycle events are broken down into different types (Creation, Dissemination, Research and Custody), with subtypes below them. In this way, events are used more as a guideline for capturing events than a way of literally representing the life a work. The main limitation to this model is that it does not model the linkages between activity types and so would be rather difficult to operationalise in its current form. Conservation, for example, may be triggered by a Dissemination event, and may itself trigger activities in Creation. This lack of connectivity reflects a problem with lifecycle models artificially viewing activities as discrete.

In software engineering the life of a software product (i.e. the output of software engineering processes) can also be understood through lifecycle models. The IEEE Software Engineering Body of Knowledge presents an overview of such models, stressing their “wide variety” (IEEE Computer Society, et al., 2014, p.8-5). They contrast two kinds of approach within the field, linear models and agile (or iterative) models, and distinguish them by the tendency of the former to be heavily specified prior to development work, while the latter involves iterative returns to requirements specification. In a similar fashion to the DOCAM model, lifecycle models are understood as being composed of processes, which they define as the “set[s] of interrelated activities that transform one or more inputs into outputs while consuming resources to accomplish the transformation” (IEEE Computer Society, et al., 2014, p.8-1). In software engineering, the idea of lifecycle model relates closely to the idea of software evolution. Indeed, the IEEE standard covering lifecycle processes defines

the software lifecycle as the “evolution of a system, product, service, project or other human-made entity from conception through retirement” (anon. ISO/IEC/IEEE Std 12207-2008: Standard for Systems and Software Engineering - Software Life Cycle Processes, 2008, p.4).

All three of the lifecycle models introduced above use process or activity types as a way of conceptualising the life of a subject entity. The DDC Lifecycle Model acknowledges the interconnectedness of lifecycle phases by sequencing them, although this sequence is of a relatively linear nature. In contrast to the DCC Lifecycle Model, neither the DOCAM model nor the IEEE lifecycle approach conceptualise the mapping out of the life of their subject as linear. Instead they take a more flexible approach with the aim of simply modelling individual lifecycle events and categorising them, without making assumptions regarding their sequencing. These kinds of model have the benefit that they make fewer assumptions about patterns of change within the life of their subject. Returning to the purpose of our analysis, a lifecycle perspective on the documentation of software-based art appears to have immediate value in that it would allow us to identify stages at which documentation should be generated or revisited. Whether the realities of change in software-based artworks can actually be represented in such a way remains unclear however.

In contrast to the lifecycle model, a continuum model offers a perspective that situates its subject as something contingent on and connected to its context. The idea of a continuum model stems from records continuum theory, a school of thought in records management and archival theory emerging in Australia in the 1990s (McKemmish, 2001) and first formalised as a model by Frank Upward (Upward, 1996, Upward, 1997). The notion of a records continuum offers an alternative to lifecycle metaphors by not specifying discrete phases within the life of a record at all, but rather conceiving of the record’s life as a continuum (i.e. a continuous sequence). In doing so, it accepts a fluidity to the identity of records, which Sue McKemmish describes as “always in a process of becoming” (McKemmish, 1994, p.8). The model also emphasises a post-custodial approach to records management, wherein the archival organisation need not have control over a record for it to engage with its care, so resisting the idea that entering a collection signals a records end-of-life and transition into an archival phase of existence.

Unlike the lifecycle models described above, the continuum model is not intended to be put directly into practice, but rather presents, as Barbara Reed puts it, a “method of thinking that challenges all archivists to engage on a broad social canvas” (Reed,

2005, p.1). This makes appraising the model rather difficult, as it has had few, if any, practical implementations. Linda J. Henry criticised records continuum theory, alongside several new theoretical trends within archiving that gained traction in the late 1990s, for having “little basis in archival theory and practice and [containing] alarmist language, unnecessary jargon, technobabble and unclear new ideas” (Henry, 1998, p.326). If the criteria is ease of comprehension, the records continuum model in the forms it has been represented so far does indeed come across as unclear—for instance, the multi-dimensional representation of the model lacks any formal definition of its axes and layers. While there is a clear gap between the theory and any kind of derivative practice, this does not necessarily make the model, and others like it, useless.

Reed acknowledges the complexity of the model, and the fact that it can be subject to multiple readings (Reed, 2005). In light of the theoretical background of the model and the comments of Reed and other champions of continuum theory, it is perhaps more pragmatic to consider the continuum model as a tool for deconstructing dogma in archiving and related fields. My own reading is that the continuum model helps us to see the object of preservation as something never definitively actualised and possessing multiple meanings for different stakeholders. Seen in this light, the processes of change that occur in the life of a software-based artwork may send ripples running through time and space that affect the meaning and identity of the artwork. By dispensing with the implied significance of lifecycle stage transitions (including that of custodial change), the model better reflects the possibility that an artwork’s life continues even within a museum environment. The use of continuum principles as the basis for a model-based approach to preservation in research as part of the PERICLES project (Lagos, et al., 2015) indicates that indeed, the continuum metaphor may hold value in practice.

Both model types discussed in this section are means of understanding complex phenomenon through simplified views. As for any such effort, it must be acknowledged that they cannot represent an objective reality but rather, an interpretation. Therefore, it is most helpful to consider not whether a lifecycle or continuum perspective is the correct one, but how lessons can be taken from both and used to guide documentation strategies. While the conservation workflow explored in Chapter 3 implies that lifecycle-like stages can be observed in conservation practice, the precise nature of change at the level of software remains unclear. In the following section I explore how processes of change might be

characterised in relation to the connections between lifecycle and continuum principles established. I draw on evidence from the artwork case studies, whose life histories have been examined and mapped based on existing documentation.

6.3. Perspectives on Software Evolution

In beginning this section, I want to consider the processes which lead up to the initial realisation of an artwork, which might conventionally be understood as relating to its *creation*. The DOCAM lifecycle model, introduced in the previous section, presents a particularly nuanced conception of the creation of a work which distinguishes this act from linked dissemination processes of installation and presentation. They define creation activities as consisting of the:

> "definition of the concepts mobilized and their method of structure (conception), definition of the presentation method, and the production of elements required for the work's presentation (materials, environmental aspects, etc.)." (DOCAM, n.d.)

For software-based art, interaction with software development processes permeates all aspects of this idea of creation. Understanding exactly what these processes were like is difficult where they occur outside the institution and prior to acquisition. The artefacts of the processes of production can help us to understand them to some extent—most significantly the source materials of the software. Software-based artwork source code for example, has been found to include significant traces of the creative process through code comments, design choices and unused code (Engel, & Wharton, 2015). In a conservation context, it is tempting to view these processes as historical actions, as such artworks are often acquired some length after they were created. The average time for the artwork case studies examined was a three-and-a-half year gap between initial production and acquisition, with the longest gap being seven years (for Cory Arcangel's *Colors*).

However, these case studies also illustrate how the nature of museum interaction with artists and art-making sometimes challenges the idea that a software-based artwork could ever fully leave the creation phase—or to use the language of continuum theory, become *actualised*. Foremost, processes which might constitute acts of creation continue to occur after acquisition as the result of ongoing realisation and occasional treatment of the works. Examples of this have occurred at numerous times for the artwork case studies. The ongoing development of the Jose Carlos Martinat Mendoza's *Brutalism* software at Tate has involved the refactoring of the Java source code on which the work was built to accommodate the use of USB printers, replacing

the obsolete DB-25 parallel port printers. The *LiMac Museum Shop* website, by Sandra Gamarra, remains under the control of the artist and is regularly updated, and so resists the idea of the artwork's stabilisation and transition into museum custody. In these cases, the emergence of the artwork extends beyond conception, beyond the first realisation and beyond even its entering the care of a museum.

In software engineering the study of the patterns of change in software programs is known as *software evolution*, or more precisely, "the process by which programs are modified and adapted to their changing environment" (Herraiz, et al., 2013, p.1:1). Meir M. Lehman's 'Laws of Software Evolution' are the most well-known theories within this field of study and constitute a set of observations that characterise the process of software evolution. These were developed and refined gradually between 1974 and 1996, driven by a growing body of research into their validity (Lehman, & Ramil, 2003). Most interesting to us is Lehman's classification of software types, which he uses as a way to understand why the laws only apply to some programs. The typology, known as the SPE scheme, was initially developed in one of his early papers in relation to software programs (Lehman, 1980) and later revised as the SPE+ scheme in reference to software systems (Cook, et al., 2006). The three types that constitute SPE+ plus can be summarised as follows:

- **Type S** (Specification-based) software can be fully defined as a complete and unchanging formal specification. The acceptability of the software to its stakeholders is contingent on whether it satisfies this specification or not. Type S software defines the conditions in which software evolution does not occur.

- **Type P** (Paradigm-based) software attempts to solve problems and maintain consistency with a particular paradigm specified by its stakeholders. The acceptability of the software is contingent on whether it successfully solves this problem and remains consistent with a paradigm to the satisfaction of its stakeholders—a process which generally involves compromise. Type P software is more likely to evolve than Type S, but this is constrained in some way to ensure the paradigm is maintained.

- **Type E** (Evolving) software interacts with the external world in some way—by design—and can never be fully specified as the software must be responsive to its environment. The acceptability of the software is therefore contingent on whether it is able to continue to respond to its changing environment and context. Type E software must evolve for its survival or otherwise become

progressively less useful to its stakeholders.

The SPE+ types are helpful in characterising the different kinds of evolutionary pattern that can be observed among software-based artworks after their creation. It should first be acknowledged however, that all software-based artworks can to some extent be considered Type E software, in that they are all embedded in the real-world through their ontological status as software performances, realised as artwork events—contingent on the environment and context in which this occurs. Indeed, Cook et al. acknowledge that true Type S software are rarely found outside of theory (Cook, et al., 2006), and suggest that both Type S and Type P software can only exist through constraints placed on the software by stakeholders. As I will illustrate below, this idea of varying degrees of constraint is helpful when we look at the different evolutionary potential among the case study artworks.

Some software-based artworks have characteristics of Type S software, in that they are highly specified, and evolution is undesirable and so constrained by those involved in the conservation of the work. John Gerrard's *Sow Farm*, for example, is a work precisely specified at the level of the software. Preserving this exact expression of the software and its consistent performance through time is therefore desirable: the positioning of each virtual building, the quality of each texture map and the precisely choreographed intensity of the simulated lighting. Breaking with Lehman's program-centric perspective, in this case the environment of the software might also be constrained using virtualisation so preventing the need for evolution to occur at the software level. The characteristics of Type S software are similar to those of Laurenson's "thickly" specified time-based media artworks (Laurenson, 2006), in that change is less acceptable for these types of artwork.

Other software-based artworks are more akin to the Type P software, in that they were created to solve a problem or implement a paradigm. For example, the software used in Rafael Lozano-Hemmer *Subtitled Public* was developed to implement the paradigm of projecting subtitles onto visitors to the exhibition space. As this paradigm is more important than the precise way it has been specified, *Subtitled Public* may have to evolve to ensure that consistency with the paradigm can be maintained when the work is realised in the future in a changing environment. The work does not engage with this changing environment by design however, and the paradigm itself is relatively well determined and can be considered in isolation. In reality, the practical challenges of realising *Subtitled Public* in a changing technical environment create a tension between the extent to which evolution might need to occur in order to maintain

the paradigm, and the degree to which there is flexibility in the paradigm itself—a tendency toward the latter in the future would indicate *Subtitled Public* is shifting toward a Type E program. Characteristics of Type P software are similar to those of Laurenson's "thinly" specified time-based media artworks (Laurenson, 2006), which allow for a degree of change in their realisation.

There are software-based artworks for which characteristics of Type E programs come to the fore. For these works, evolution is an inherent part of their identity. For example, Jose Carlos Martinat Mendoza's *Brutalism* software harvests search results for a particular term from the Google Search API, which are accumulated in a database and printed in the gallery. Thus, part of its identity lies in interaction with this changing external API and the activity on the internet that feeds Google's search algorithms, resulting in emergent meaning in the text harvested. When realised, *Brutalism* becomes part of a wider socio-technical environment, extending beyond the boundaries of the exhibition space and into the external world. In the case of *Brutalism*, the software can never be fully specified nor understood in relation to a fixed problem, as its realisation is tied to the changing properties of external environment and context. The software must also therefore continue to evolve in order to maintain this connection. If this was found to be impossible at any stage, and the artwork disconnected from this context so that it no longer accumulates words, it would shift more towards Type P software.

There is of course some distance between the kinds of embedded, continuity-driven software systems with which software engineering largely concerns itself, and the software-based artwork as something realised and thus ephemeral. Even for works which require persistent availability, such as the *LiMac* website, change does not appear to happen at a single constant pace (I examine this case study in more detail in later in this chapter). For software-based artworks that enter collections, software evolution in response to changing technical environment seems to occur as a kind *punctuated equilibrium*[22]. This term extends the biological evolution metaphor, and references evolution patterns which involve long periods of relatively slow change (or stasis), punctuated by periods of rapid evolution. The trigger for these rapid evolutionary events, in the case of the artwork case studies examined, appears to

[22] This term has its origins in a paper by Niles Eldridge and Stephen Jay Gould (Gould, et al., 1972), which contrasts their theory of "punctuated equilibria" with the traditional model of biological evolution, "phyletic gradualism", which views change as steady and continuous.

vary depending on the type of program but be heavily associated with the revisiting of works in the context of display. In the future, as the technology involved in these case studies ages (most were produced in the 2000s), we can expect preventative conservation efforts to become a second major trigger.

In this section I have developed an overarching theoretical framework for understanding how software evolution occurs for software-based artworks, which helps to explain how patterns of change may differ between artworks. Major evolutionary events would trigger a return to existing documentation: the relevant body of significant knowledge would be reconsidered, and technical documentation revisited to align it with the resulting software structure. We are also interested in documenting the processes of change themselves, however, as this ensures a provenance trail and provides records of processes to allow the reconstruction of the artwork's technical history. This relates to the well-established ethical codes guiding the conservator in ensuring treatments and conservation measures are documented (The Institute of Conservation, 2014, American Institute for Conservation of Historic and Artistic Works, 1994). Documentation of the process might be addressed at different levels of abstraction. At a high level, this might be understood as relating to the goals of the process and the production of new versions of software and artwork. These I refer to as macro-level change patterns. At a low level it would relate to the formal material (understood in relation to Kirschenbaum's theory of materiality) manipulated by the artist and collaborators. While this might be most obviously understood as the shaping of code, as I have demonstrated elsewhere in this thesis, this work often involves other formal materialities such as software interfaces and production tools. These I refer to as micro-level change patterns. Micro-level change is often necessary to achieve macro-level change, while macro-level changes may result from micro-level change—the two pattern types are inherently linked. The utility of the distinction is that it distinguishes between two levels at which change can be documented, even where the same change processes are being observed. I discuss each of these levels in more detail in the next two sections.

6.3.1. Macro-level Change Patterns

At the macro-level we can observe the transformation between software versions, among which certain kinds of transformation occur repeatedly. The terminology for describing these is not well developed in the context of software-based art, but terminology from software engineering can be repurposed to help fill this gap. Below I propose a set of process descriptors for the various kinds of software transformation

which might occur during the evolution of a software-based artwork (including examples from case studies where possible), in relation to activity engaging the source and executable code representations:

- **Refactoring:** Software is revised to improve or correct non-functional attributes, without altering its functional attributes, within the same environment as the original. In the case of *Subtitled Public*, the Delphi software was refactored to allow use of higher resolution camera feeds.

- **Rewriting:** Source code is rewritten to add or alter functional attributes, within the same environment as the original. Reengineering, redesign and rearchitecting are terms used to indicate similar kinds of change in software development, and in practice may be hard to distinguish from rewriting. In the case of *Brutalism* the original Java code has been partially rewritten to allow the use of USB printers instead of DB-25 parallel port printers, which required an entirely new Java package.

- **Migration:** Source code is rewritten for a different operating environment or platform without impacting its functional attributes. This is widely understood within digital preservation in relation to format migration, but is also significant in software development where it is sometimes referred to as *porting*. The Shockwave version (including its Lingo scripting) used in the 2003 version of Becoming was migrated to Flash (and ActionScript scripting) for the 2010 version, as an experimental conservation treatment aiming to maintain the software's functional and non-functional attributes.

- **Compilation:** Source code (and other material) is compiled into an executable representation. This process occurs for almost all software-based artworks during their creation, although for some languages (e.g. Java) it is to an intermediate representation which must be interpreted to be executed. Compilation can be carried out with different parameters, which can be captured using build logs or metadata. The form and availability of these is dependent on the programming language and development environment used.

- **Decompilation:** Executable representation is transformed into higher level code approximating the source code. I present experiments with this transformation technique in Chapter 4, and propose it is likely to be a useful

tool in conservation practice for dealing with missing or inaccessible source code.

- **Encapsulation:** Binary representation is captured with elements of its software environment and configuration (for example, as a disk image), in order to allow effective virtualisation or emulation. This method is being used in the preservation of software-based artworks at Tate and was used extensively during this research to create controlled environments for the examination of software.

The adoption of such a vocabulary to describe software-based artwork transformations provides one foundation for documenting macro-level change and could be accompanied by a more detailed description of the process, its purpose and its justification. One potential usage relates to the practice of production history documentation at Tate. The change history of media elements relating to an artwork, are captured using what is known as a “production diagram”, a representation of their lineage presented as a branching directed graph. An example for *Becoming* (2003) by Michael Craig-Martin is embedded below in Figure 19. The terminology introduced above presents a vocabulary with which relationships between software media elements might be described within this framework.

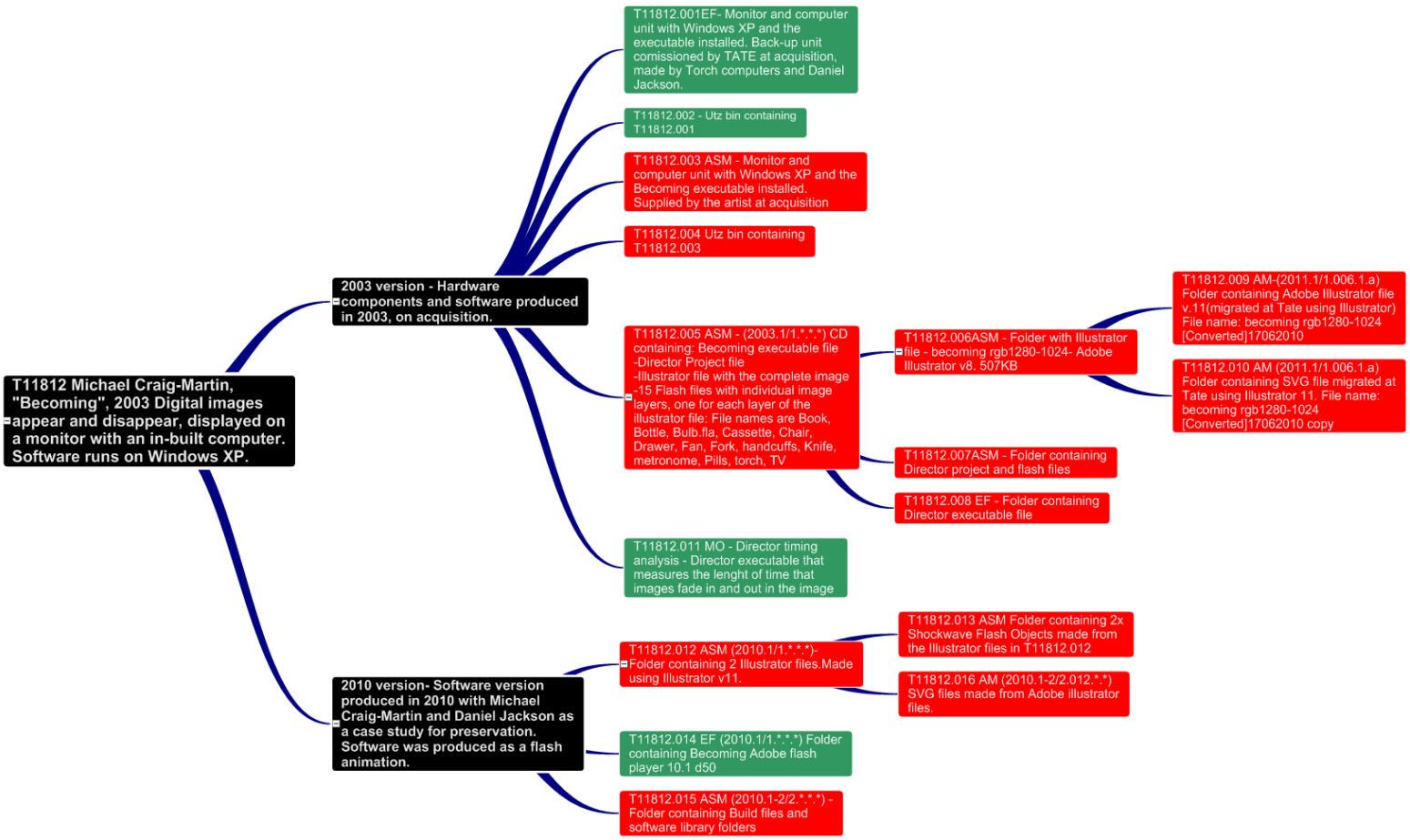


Figure 19. Production diagram for the artwork *Becoming* by Michael Craig-Martin, created by the time-based media conservation team at Tate. Black boxes indicates information (not corresponding to an actual component), green indicates a media component suitable for exhibition use, while red indicates an archival media component not suitable for exhibition use.

In commercial software development, transformations would also be reflected in software versioning: the practice of uniquely identifying the representations of a program produced by development processes. This usually takes the form of an incremental version number for technical users, but versions can also be expressed through non-technical language aimed at consumers of software products. For example, Apple's naming scheme for its OS X operating systems uses memorable themed names alongside more granular version numbers. OS X 10.10 is known as "Yosemite", as part of series of version names themed on Californian landmarks, but in fact has six patch versions (10.10-10.10.5) and many more builds (identified using a separate number) which reflect the implementation of various bug fixes.

This distinction between technical and public facing versioning schemes is reflected in software-based art. For instance, there has only been one version of *Subtitled Public* the artwork. However, at the software level, there have been multiple minor versions generated which either improve or slightly modify the program's behaviour. In addition, there have been multiple realisations of the work for different exhibitions, which provides a third potential understanding of version. The artist David Rokeby for instance, considers versions in this way, relating them to an iterative process of technical development—such as the evolution of the five different realisations of the ongoing work *The Giver of Names* (Rokeby, 2010). With this possibility in mind, there is a need for clear language with which to make distinct the various artwork versions, artwork realisations and software versions. I return to this later in this chapter (see Section 6.4) but for now consider only the software level. Granular versioning of software is advantageous for conservators in the same way that it is advantageous to unambiguously identify any resource: it can be located reliably, referenced unambiguously and its relationship with the artwork and its realisations recorded.

There is no single standard approach to version numbering, but Tom Preston-Warner's "Semantic Versioning" scheme (I here reference version 2.0.0) is widely adopted and understood (Preston-Werner, 2013). Versions are represented using three numbers in the form "MAJOR.MINOR.PATCH" (e.g. 2.10.5 represents major version 2, minor version 10, patch 5), which are incremented to indicate different levels of change. The major number is incremented where a change has been made which breaks backwards-compatibility. The minor number is incremented where the changes have less impact and add functionality in a way which is backward-compatible. The patch number indicates least impact and is incremented when a change is a backwards-compatible bug fix. When a particular number is incremented,

this resets the numbers to the right of them, which can then be incremented from zero again.

While useful as a model, in practice the practicality of this kind of formal approach has limitations. Foremost, clarity of versioning comes with a contingency on the methods of whoever is making the changes. The reality is that different programmers will want to work in different ways, and development might not occur in such a way that permits clear identification of the meaning of a change. For instance, during the rewriting of the *Brutalism* software in the run up to the artwork's display at Tate Modern, numerous versions of the software were generated in a short space of time in order to rapidly test prototypes of the modified software. Reconstructing their relationship with the source codebase and the evolving work is now difficult. Allowing such flexibility in working methods however, may be essential within these collaborations. As I will demonstrate in the following section, micro-level change tracking (which is largely systems-driven) is another way in which this problem can be managed.

6.3.2. Micro-level Change Patterns

At the micro-level, change can be understood not through software transformations, but as gradual change at the level of the material (understood in relation to Kirschenbaum's formal materiality, introduced in Chapter 2) manipulated in the realisation of the work. This material might be source code, or a development project manipulated via interfaces. Source code changes at this level may be understood from documentation generated by systems that interact with the process directly, or from retroactive inference using available artefacts, which I will discuss below in turn. As such, documentation of this kind of change is generally contingent on the availability of source code and associated artefacts, on which I will focus in this section.

Software development at scale (be that understood as a large codebase or numerous collaborators) necessitates the orchestration of many individuals working on a code-base simultaneously. As a consequence of these challenges, low-level change management systems have emerged to support modern software development practice. Source code version control systems (VCS) allow the tracking of multiple, parallel modifications at the source code level through a system of access control and change tracking (Yuill, 2008). Managing changes to source code is an important activity in software development, particularly in ensuring multiple programmers can

work without conflict on a complex code base. There are a multitude of VCS platforms. Some involve the use of a single centralised repository such as Apache Subversion (or SVN), while others such as Git (perhaps the most widely known through the popular cloud-based service GitHub) use a distributed method involving a local repository and committing changes to a shared repository as a separate step. An example of commit record, taken from the open-source GitHub repository for the Rafael Lozano-Hemmer artwork *Level of Confidence* (2015), is shown in Figure 20 below.

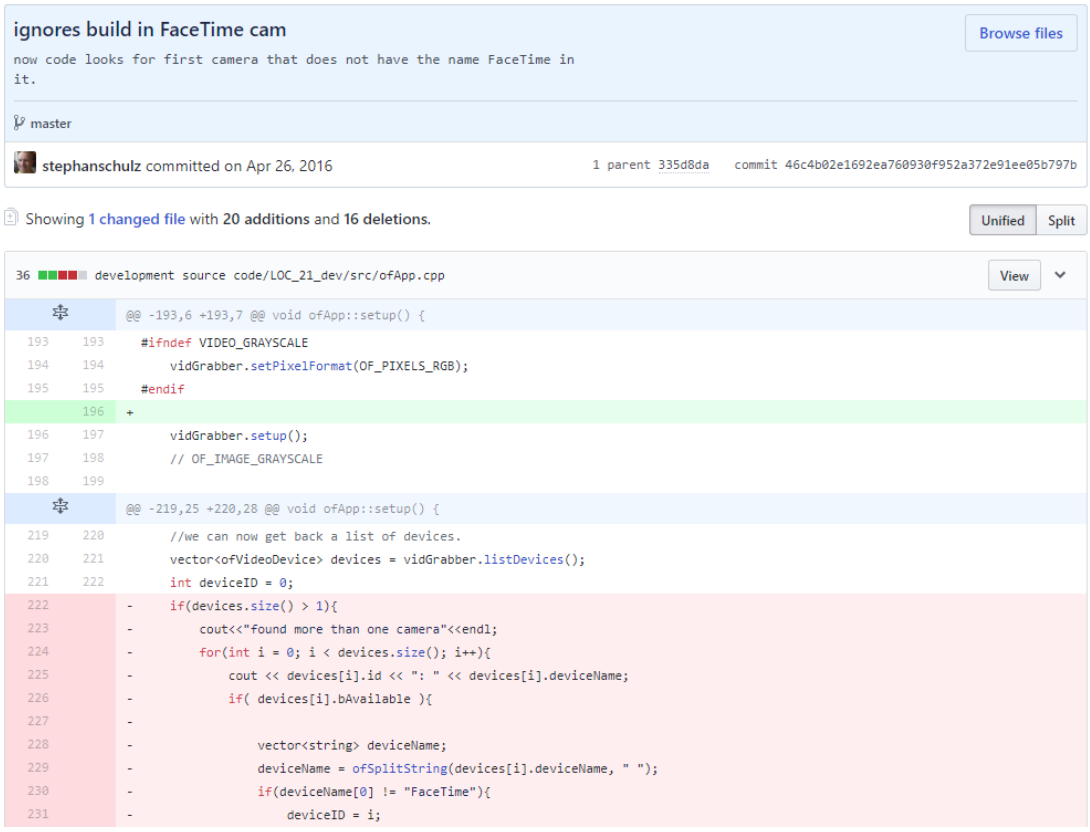


Figure 20. Screenshot of a record of a C++ code change committed to a GitHub code repository for the Rafael Lozano-Hemmer artwork *Level of Confidence* (2015), by programmer Stephan Schulz. The commit record includes metadata about the author and date, a description of the change, and a visual indication of the changes made to the code itself (green lines have been added, while red have been removed).

As a by-product of the process of version control, a VCS may provide a comprehensive record of alterations to the code (including by whom alteration were made) and by inference, of the development process. Therefore, where these systems have been used, they have great potential interest in the study of software-based artworks. Where they can be used during future development work for a

software-based artwork, they may provide a rich addition to the documentation of a work. This is already occurring in museum conservation practice. Ian Cheng's *Emissaries* series of three software-based 3D simulation artworks were exhibited at MoMA PS1 in 2017. During the display of these works, the artist worked from a version control system in the care of the museum (B. Fino-Radin, personal communication, 17 June 2016). According to Ben Fino-Radin, then a Media Conservator at MoMA, this allowed him to update the software during the exhibition to fix bugs, and then later allowed MoMA to acquire the VCS as a documentary record of the works development, as all the changes made are represented within this system. At San Francisco Museum of Modern Art (SFMOMA) in 2015, media conservator Martina Haidvogl worked closely with the artist Jürg Lehni on the acquisition of his robotic drawing machine *Viktor* (Haidvogl, 2015). Again, the artist worked from a Git repository (in this case using Bitbucket, a cloud-based Git platform) which could be synced with a computer at SFMOMA and archived. The code repository included a record of the various code modules developed, automatic syncing with any changes occurring during the acquisition process, and even the tracking of documentation authored in Markdown.

Where VCS or other system-based change management tools have not been used, there may be other ways to infer information about the development process. For example, if more than one version of the source code is available, automated methods can be used to compare them. In the case of *Colors*, there is more than one version of the work—the 2005 version which was acquired by Tate, and a 2009 version released as free and open source software called *Colors Personal Edition* (Arcangel, 2017). While the actual functionality of the software is very similar for both versions, the *Personal Edition* is open source and distributed over the internet for free. This version allows the user of the software to process a video file of their choosing. The 2005 version of the work, on the other hand, is intended to specifically play through Dennis Hopper's 1998 film (also titled *Colors*) 404 times (as this is the number of lines in the video provided), and is to be projected in an exhibition setting.

Clearly these two artworks share a similar core concept (playing back a video file line by line), while also being guided by slightly different intentions and modes of presentation. Source code analysis allows us to compare on a technical level how similar the two pieces are. An automated line-by-line comparison tool (known as a diff tool) allows us to reveal that the code used is identical bar one change—as illustrated in Figure 21.

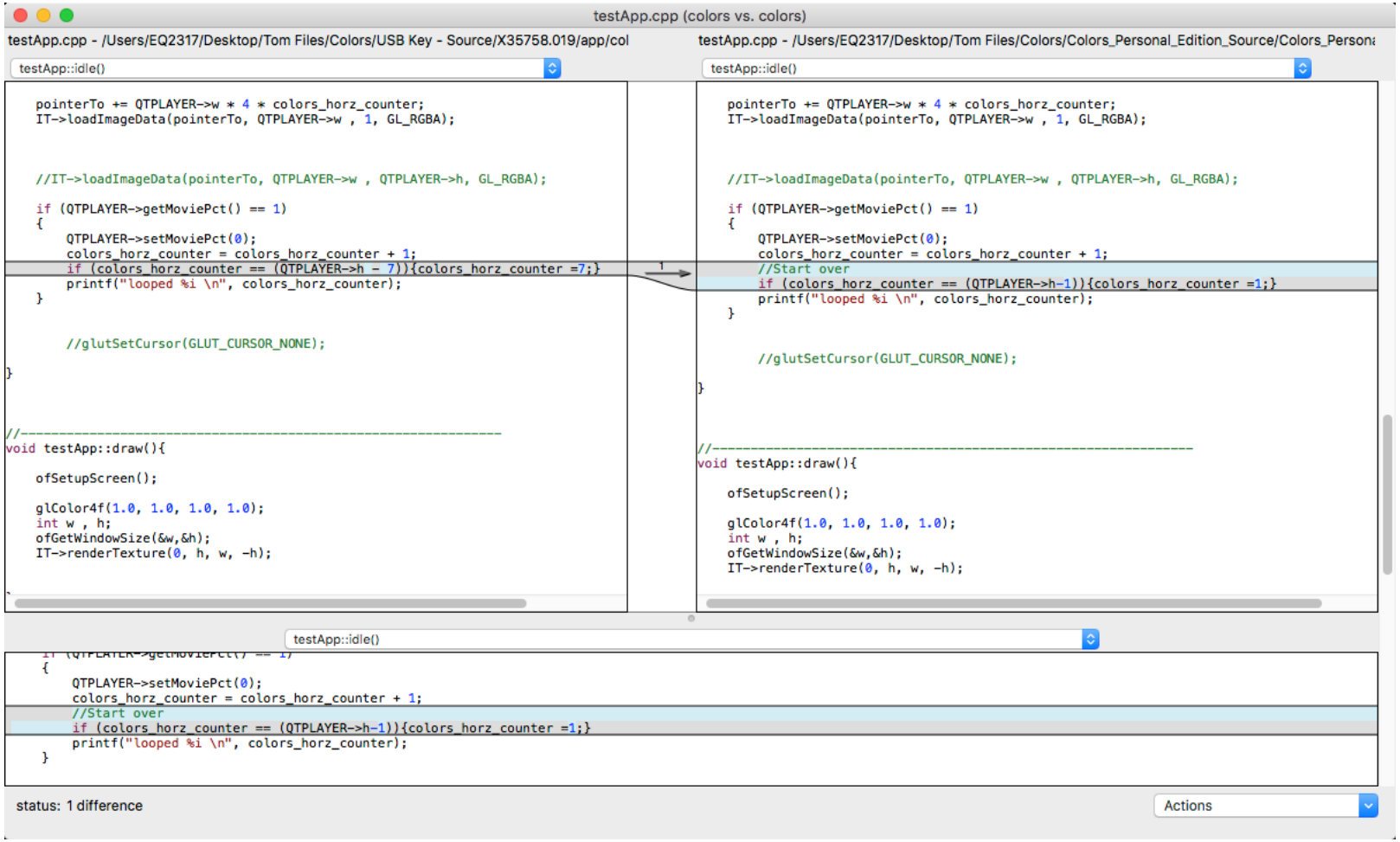


Figure 21. Results of an automated code comparison between the source code of Cory Arcangel's *Colors (*2005) (left) and *Colors Personal Edition* (2009) (right), using the FileMerge (Apple, 2016a) tool package with XCode 7 (Apple, 2016b).

The first line of the change is a code comment (i.e. non-functional text embedded in the code structure), the reasons for the removal of which are unclear, while the second line skips a certain number of pixels in the 2005 version in order to avoid processing the black letter-boxing of the source video file (processing it would result in a black screen rather than bands of colour for a portion of the software performance). In this case, understanding code level change is much less important than understanding the artist's own versioning systems. This is an interesting technical art historical insight however, and in this case further emphasises Arcangel's elevation of the concept over the technical implementation.

In addition to code-level change and processes of software development, there is a need for systems-based change documentation for technical environments and their representations (e.g. disk images and virtual appliances). Where encapsulation methods are used to create emulated and virtualised representations of software-based artworks, a record of the changes required to create a suitable execution environment would be valuable (an idea introduced in the context of the reconstructive workflow described in Chapter 4). One approach would be to use manual techniques to carry out the systematic capture of environment information at key points in time such as acquisition or after a treatment. Although approaches to continuous monitoring (and documentation) of environment information have been developed for digital preservation purposes (Corubolo, et al., 2014), there remain practical obstacles to their integration. Such tools involve invasive embedding in the host system followed by continuous operation which may be resource intensive and put strain on the relationship with artist or programmer. While the approach highlighted is modular and thus extensible as new technology arises, there are also currently limits on the extent of the system environment they are able to monitor.

Managing change within disk images and virtualised or emulated environments is in many ways similar to managing complete computer systems, but also offers a means of achieving a higher level of environment monitoring and capture. A number of emulation and virtualisation software packages (e.g. QEMU, VirtualBox, Vmware) are able to utilise a technique called *copy-on-write* to automatically document changes made to read-only disk images during operation. This involves use of a secondary disk image format (for QEMU this is in the qcow format, for example) which will only grow in size in order to store modifications (not a complete representation) made to an underlying disk image, rather than writing directly over the data in the base image. If data is requested from the base image, it will be retrieved directly from there, while

if it is requested from modified sectors, it will use the secondary image.

6.4. Representing Versions in Information Systems

The interlinked set of entities which are the result of the processes of software evolution described in the previous section will, much like the software structures discussed in Chapter 4, require a model through which they can be represented in information systems. This provides a means of connecting a particular software structure (including its physical and digital constituents) with the concepts that give it meaning: the artwork, its realisations and its versions. It also provides a consistent framework for connecting conservation activities with the appropriate entity in relation to the ongoing life of the work. Having an appropriate conceptual model is the first step in ensuring that we can accurately record this information. The software-based art domain has received little attention in the definition of such models. In this section, I explore the application of a mature model from the libraries and archives domain to this problem.

The Functional Requirements for Bibliographic Records (FRBR) was developed to provide a structural model for relating information contained in bibliographic records to the needs of users, and ultimately improve the efficiency of finding, identifying and accessing bibliographic records (IFLA Study Group on the Functional Requirements for Bibliographic Records, 2009). While originally designed for the description of bibliographic materials, the model has been influential beyond and has already been explored in relation to the description of software. Matthews et al. develop an interpretation of the model for the description of software systems (Matthews, et al., 2010), the focus of which is primarily on software products and terminologically divergent from an art use case. McDonough et al. apply the model (as-is) to a work of interactive fiction (expressed as software) with a complex, branching version history, and find it suitable for describing its many manifestations with relative clarity (McDonough, et al., 2010). The DOCAM project also applied the FRBR model as-is, in this case to the hierarchical description of media art (DOCAM, n.d.). An additional level of granularity below item called “component” is proposed, which serves to capture the parts of an “item”.

In Table 7 below I compare the bibliographic IFLA version of FRBR (as used by McDonough et al.) to the model developed by Matthews et al., and in the final column, propose a set of entities with which to describe software-based artworks and their linked representations.

FRBR Standard (IFLA 2009)		Conceptual Model for Software (Matthews et al. 2010)		Conceptual Model for Software-based Art	
Entity	**Description**	**Entity**	**Description**	**Entity**	**Description**
Work	A distinct intellectual or artistic creation	Product	The product is the whole top-level entity of the system, and is how the system may be commonly or informally referred to.	Artwork	A distinct intellectual or artistic creation.
Expression	The intellectual or artistic realisation of a work	Version	A version of a software product is an expression of the product which provides a single coherent presentation of the product with a well defined functionality and behaviour.	Version	An expression of the artwork with well defined formal, functional and behavioural characteristics.
Manifestation	The physical embodiment of an expression of a work	Variant	Versions may have a number of different variations to accommodate different operating environments.	Variant	A specific implementation of a version which has broadly similar formal, functional and behavioural characteristics.
Item	A single exemplar of a manifestation.	Instance	An actual physical instance of a software product which is to be found on a particular machine is known as an Instance.	Realisation	An embodiment of a particular variant of the work in time and space.

Table 7. Mapping of the IFLA FRBR model (IFLA Study Group on the Functional Requirements for Bibliographic Records, 2009), FRBR-based Conceptual Model for Software (Matthews, et al., 2010) and an FRBR-based model for describing software-based artworks.

A significant limitation in representing a software-based artwork using the FRBR and Software models is that, emerging from bibliography and software "product" preservation respectively, they describe (at the lowest level) singular instances understood as discrete objects—"Item" and "Instance" respectively. The notion of "realisation" as it has been developed in this thesis cannot be understood as a discrete object. A realisation is not physically (or digitally) persistent through time, rather it is essentially an event, often understood in relation to the coming together of many components (i.e. the physically and digitally persistent parts of the artwork which are stored and managed even when an artwork is not realised). I reject the DOCAM proposal of a "component" level, as the structural complexity at this level would be difficult to represent in a useful form. Instead, the realisation level of the model could be connected to a representation of the software structure, as described using the conceptual model introduced in Section 4.6.3.

In Figure 22 below, the version lineage of *Becoming* has been modelled as an RDF/XML format OWL 2 (World Wide Web Consortium, 2012) ontology, developed in Protege 5.2 (Stanford Center for Biomedical Informatics Research, 2016). The classes and properties that constitute this component of the model are incorporated into the larger Software-based Artwork Structure Ontology (SASO) introduced in Chapter 4 and detailed in Appendix II.

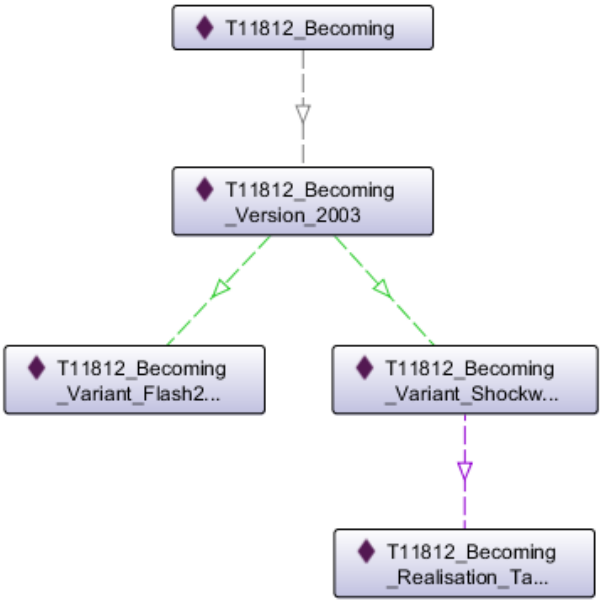


Figure 22. Representation of class instances that make up the artwork version history of *Becoming* by Michael Craig-Martin, using the SASO model. Relationships between classes are modelled as object properties, indicated by arrows (grey: hasVersion; green: hasVariant; purple: hasRealisation).

Examining the technical histories of the software-based artwork case studies, it is clear that versions—as defined in the conceptual model introduced here—are largely absent from their production histories. That is, there have been few cases where an artwork has involved more than one formally, functionally or behaviourally distinct expression, that could be considered to still constitute the same work. The only example for which this is apparent is *LiMac Museum Shop*, which I explore in depth in the next section.

6.5. Case Study: The Evolution of LiMac Museum Shop

LiMac Museum Shop (2005) is an artwork by the artist Sandra Gamarra which is indicative of some of the challenges in documenting the evolving software-based artwork. It is important to note that the work is not a software-based artwork *per se* however, as the website which might be characterised as such has a complex relationship with the artwork acquired by Tate. *LiMac Museum Shop* itself is a variably formulated installation and part of a larger body of work produced by Gamarra which is structured around the fictional "Museo de Arte Contemporáneo de Lima" ("Museum of Contemporary Art of Lima" in English). Addressing the absence of such an institution in Peru, the artist (herself Peruvian) has constructed a complete corporate identity for the museum complete with a collection, exhibitions programme and website. *LiMac Museum Shop* is one physical embodiment of the museum, and mimics the trappings of the museum gift shop, consisting of a central cabinet filled with souvenirs, many of which are branded with the LiMac identity (see Figure 23 below).

Figure 23. Sandra Gamarra, *LiMac Museum Shop*, 2005, installed at Tate Modern in 2011. The terminal providing access to the website is visible on the side of the cabinet in the right hand image.

The website has typically been presented as a part of this installation, usually through a terminal embedded in the cabinet with which visitors can interact in order to browse its content. The external manifestation of the museum that this website indicates could be seen to further enhance the illusion of the museums existence and authority. Indeed, the website continues to exist independently of the work. While *LiMac Museum Shop* was acquired by Tate in 2011, the website itself has remained hosted by the artist, while Tate Information Systems work with the artists team to acquire regular snapshots of the server data. This has allowed Gamarra the freedom to continue developing and updating the site, with only minimal requirements for the negotiation of institutional information systems. If we look at the evolution of the website, we can observe the pattern of punctuated equilibrium which I developed earlier in this chapter. There are several points of significant macro-level change where the website is redesigned and transitions to a new technical platform. The visual characteristics of these changes are illustrated in Figure 24 to 26 below.

Figure 24. Screenshot of the front page of the static version of the LiMac website, which was live from 2005-2007. © Sandra Gamarra 2018.

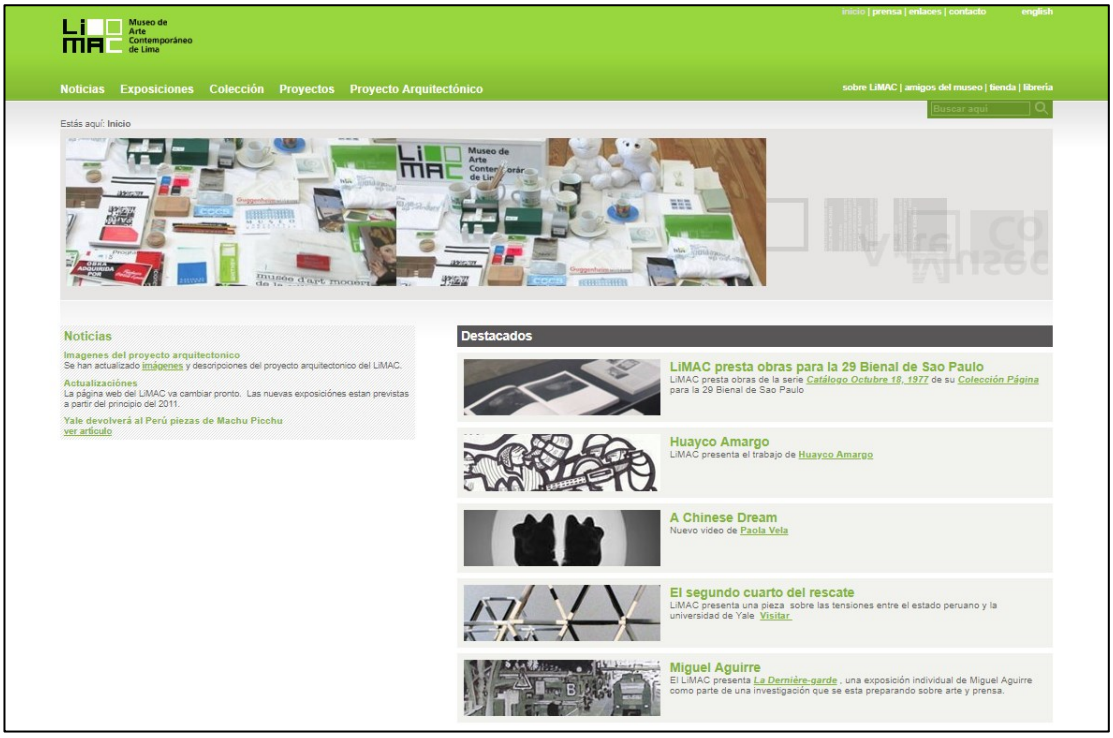


Figure 25. Screenshot of the front page of the MODx version of the LiMac website, which was live from 2007-2012. © Sandra Gamarra 2018.

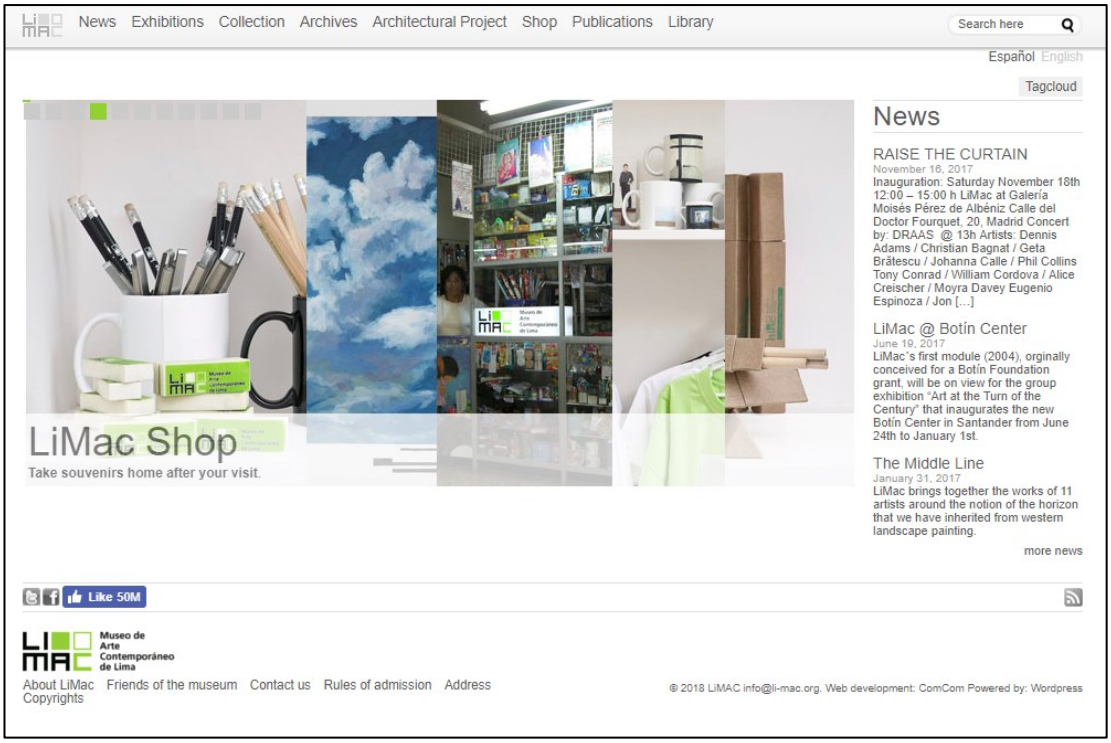


Figure 26. Screenshot of the front page of the WordPress version of the LiMac website, which has been live from 2005-present. © Sandra Gamarra 2018.

The underlying technical platform used is representative of popular web technologies through time, moving from static HTML pages with embedded Shockwave elements, to a content management system (CMS) MODx, followed by a later transition to the more popular CMS WordPress. The past forms are an important part of the technical history of *LiMac Museum Shop*, which would have incorporated different versions of the site depending on the time of realisation. Pip Laurenson points out that in using the tropes of the museum website, the LiMac website "is not only designed to evolve and change over time but [...] also references a form for which this is to be expected" (Laurenson, 2013, p.88). A record of these forms is therefore core to establishing the artwork's link with museum branding and web design as it too has evolved through time. Throughout the website's history, micro-level change has occurred in a more regular pattern through the addition of content, such as new publications and additions to the collection. There is value in recording these incremental updates, which present a record of the artist's engagement with the site through time, and allow a more direct link to be established between any one realisation and the state of the site at a particular time.

Several complementary technical options might be considered for the documentation of the website, which overlap with the preservation of the software itself. The first is

to simply save snapshots of the website server stack (including all the sites back-end data and supporting software) at different points in time. This would represent the most complete capture of the site, but would also demand the most storage space, so potentially preventing regular capture. The second is to capture the site through web crawls or other website archiving tools. These capture the performance from the perspective of a user agent (e.g. browser), and so would not fully capture the back-end components. However, the crawl data would require considerably less storage volume as a result, and an automated crawl (using the Internet Archive's Heritrix crawler for example) could for be used to capture daily snapshots. The third would be to use a version control system to monitor changes at the code level. This would ensure that the actual systematic change observed at the level of code is captured. This would not fully capture the nature of change at the content-level however, much of which would be stored in a SQL database.

The future of the artwork may lie in the artefacts generated by the first two processes, as these provide a means of reconstructing a moment in the website's history. While *LiMac Museum Shop* as an installation that has become relatively fixed in terms of its material constituents (the souvenirs and the cabinet itself have been collected by Tate), the website has continued to evolve. The point at which *LiMac Museum Shop* entered the Tate collection marks a branching point in its history, when a historical version of the website, built in MODx, was created on Tate servers—though not made publicly accessible. In agreement with the artist, a live version of the site will be shown as part of the installation, while the process of capturing historical versions continues as the site evolves. However, as the installation itself is now fixed, and visually and thematically linked to a particular period of the LiMac project (spanning from 2005 to the last time it was shown in 2011), there may come a point in time when presenting an older snapshot of the site may be the most appropriate choice. This would then detach the site as seen in the installation from its original context, and leave it out of sync with the ongoing LiMac project and the evolving web. If the MODx version, which has been isolated from the evolutionary pressures which applied to the original, were to be displayed, it would be important to convey the reasoning behind this and the installations relationship to Gamarra's ongoing practice. If museums such as Tate are to be able to effectively convey the history of evolutionary software-based works such as LiMac, there is a need to develop novel methods for doing so.

6.6. Software-based Artwork Biographies in Conservation

The earlier sections of this chapter have illustrated that change occurs both as micro-level process, where code and other digital materials are shaped and reshaped in a digital environment, and as macro-level process, where transformative events such as realisation and conservation treatment generate new identifiable versions. So far I have focused primarily on technically-driven approaches to understanding and capturing these kinds of process. These are important considerations in a museum conservation context, but understanding the meaning and significance of changes in relation to an artwork's identity requires more than tracking processes and recording transformation in information systems. The data stored in a Git repository may be a complete representation of change a technical level sense, but how do we ensure it retains meaning in relation to the artwork itself and the forces that have shaped the pattern of branches and commits? There is therefore a need for a framework capable of linking together technical documentation and the broader social and historical context of the artwork, in order to be able to effectively capture narratives of software-based artwork evolution.

The contributions of the theory of object biography appear to broadly align with these needs, particularly in offering an outlook which considers objects as products of shifting social context. The origins of this idea are found in a paper by cultural anthropologist Igor Kopytoff (Kopytoff, 1986), which proposed that we might ask questions of objects (or "things" more generally) that are similar to those we might ask of persons. Who made it and why? How has it changed through its lifespan? These foundations have gone on to inspire the development of related theory in a number of fields, including conservation. In research emerging from Dutch contemporary art conservation research project "New Strategies in the Conservation of Contemporary Art" (van de Vall, et al., 2011), the group of researchers involved introduce the idea of a biographic approach to documentation in response to the complexity and multiplicity observed in the life of an artwork. They propose that:

> "the meaning of an object and the effects it has on people and events may change during its existence, due to changes in its physical state, use, and social, cultural and historical context. The concept of the biography enables us to describe – and thereby construct – the artworks' 'lives' as individual trajectories that nevertheless may show similar phases and patterns of change." (van de Vall, et al., 2011, p.3)

These biographies, the authors suggest, need not begin or end with the acquisition of a work by a museum, and will likely need to be rewritten repeatedly or exist as "various singular interweaving partial biographies with different beginnings,

itineraries, dynamics and endings" (van de Vall, et al., 2011, p.6). How then, might such a biographical approach take shape for software-based artworks and how might it help us describe their technical histories? In the next section I consider how an approach might be developed through linkages to ideas from continuum theory introduced earlier in this chapter, which provide a mode of enquiry through which to develop biographies. I then look at a significant yet currently unanswered question of the biographical approach developed by van de Vall et al.: how it might transition from theory and research into the everyday practice of a conservator.

6.6.1. Continuum Approach to Artwork Biography

I propose that the biography of a software-based artwork might be understood in relation to ideas from records continuum theory. This outlook bears a striking resemblance to the notion of artwork biographies introduced in the previous section, in that it accommodates the artist's continued stake and involvement in the artwork's future, the multiplicity of perspectives involved in conservation, and the dynamic organisational and social forces that artworks are subject to on their evolutionary trajectory. As suggested at the beginning of this chapter, the continuum model serves more as a tool for directing enquiry than an easily repurposable approach. While methods for formally applying continuum theory are lacking, I draw inspiration from Barbara Reed's practical examples of "recordkeeping stories" to structure my approach (Reed, 2005).

A continuum-situated biographical approach, much like the records model, might consider the artwork in relation to several *dimensions*—as developed by Frank Upward in the original formalisation of the model (Upward, 1996). These dimensions are (metaphorical) spaces which the work simultaneously occupies, and serve to organise the various forces that shape the artworks life. Events and processes of change send ripple effects through these dimensions, potentially affecting others. A reinterpretation of the original records continuum model dimension set is proposed in Table 8. The term "event-process" is used to refer to those events and processes (the two are here considered as indistinguishable) which occur within the continuum, and may occur over any interval of time ranging from seconds to centuries.

Dimension	Scope	Event-Process Examples
1. Create	Relates to acts of conceptualisation, creation and	• An artwork is conceived of as an idea by an artist • An artwork is reinterpreted by an artist

	modification of the artwork (as something both abstract and concrete).	• Inherent vice results in breakdown of a material component • Source code is written • Software is compiled from source code • A version of an artwork is realised in time and space • A display computer is constructed • Decompiling compiled software
2. Capture	Relates to the formalisation (i.e. transformation into a formal model) of an artwork for a particular purpose, in order to allow some particular use, realisation or representation.	• Artwork is formalised as a set of requirements • An installation is documented • A metadata record is composed • An exhibition catalogue is published • A disk image of the original hard disk is captured • Reverse engineering documentation from software
3. Organise	Relates to modes of operation, policy and business rules within a collection, institution or other group with custodial responsibility for the artwork.	• An artwork's ownership changes • A new institutional mandate results in the need to present artworks online • A loan is requested • A new metadata schema is defined and implemented in a collections management system • A long-term web hosting agreement is drafted
4. Pluralise	Relates to the interaction of society, politics and a broader human context with the artwork.	• A semantic shift in the meaning of a conceptually significant component of the artwork occurs • A technology company goes out of business and stops producing and supporting a software product • A technology becomes associated with strongly negative connotations e.g. through criminal use of their products • A technology becomes seen as common-place or archaic • An art movement becomes a taught part of art history

Table 8. Dimensions of a continuum-based understanding of software-based artwork change, from the perspective of a time-based media conservator. Dimension numbers do not imply an increasing scale or any other ordinal arrangement.

While event-processes have been provided as examples occurring within particular dimensions, they are best understood through connections with other event-processes—so forming trajectories through the dimensions of the model. In order to illustrate its use in practice, I will present biographical fragments—or short narratives—of two case study artworks, both of which sought to engage all the dimensions of the model. The occurrence of a particular dimension is annotated within the text. In the first, I develop a biographical fragment relating to the creation of John Gerrard's *Sow Farm*, and the legacy of the choices made during development:

> *Sow Farm* was developed around the year 2009 using Quest3D (dimension 1), a 3D development environment available at the time of production that was typically used for architectural visualisation and real-time 3D simulation (dimension 4). This software was used by other artists around this time period (e.g. Samyn, 2008), which reflects an emerging interest in easy-to-use technical solutions for 3D production among creative communities (dimension 4). Gerrard worked with a team of collaborators based at a production studio in Vienna to produce the work, the process of which resulted in a number of production artefacts including documentation (dimension 2). Later in Gerrard's career, the availability of other more advanced 3D software tools (dimension 4) resulted in changes in his team's production process, and Unigine is now used as their primary 3D production software (dimension 3). While these shifts were occurring, *Sow Farm* work has been acquired by Tate in 2014 (dimension 3), and a new realisation of the work created at Tate Britain (dimension 1) and re-formalised as additional documentation (dimension 2). Quest3D has since been retired as a commercial product by its developers, in favour of supporting their new software (dimension 3) and as a result of market pressures to keep up with technological developments (dimension 4). The lack of availability of source materials (dimension 3) and software to read them (dimension 4) results in difficulties carrying out complete documentation of the work by conservators (dimension 2). The artist indicates that they would like future realisations of *Sow Farm* to remain faithful to the original Quest3D implementation (dimension 1), a preference which is documented by Tate and so provides further formalisation of the work (dimension 2), while his studio offers to provide support as a service (dimension 3).

The advantage of the approach in this case is that it highlights connectivity between processes and events in the life of the work by situating creation and production choices within a wider sociotechnical context. Particularly significant is the clarity gained over the moment at which the identity of the work becomes further fixed, as distance grows from the original production process. It also clearly identifies the way in which technological shifts in commercial 3D rendering technology directly relate to

the demands of Tate's ongoing engagement with the work and its conservation.

A biographical fragment relating to the context and acquisition of Jose Carlos Martinat Menzoda's *Brutalism* offers further illustration of the approach:

> *Brutalism* was created in 2007 (dimension 1), and in a conceptual sense draws on two historical currents: the Fujimori presidency in Peru, during which the brutalist style Pentagonito building housed the military secret service and acts of violence associated with the regime (dimension 4), and the multiple meanings of the word brutalism (dimension 4). These references link directly to the production choices made in the creation of the work (dimension 1)—the sculptural element as a scale model of the Pentagonito, and the web search and printing system as a means of deriving semi-random associations of meaning. The artwork was purchase by Tate in 2010 (dimension 3), setting a cascade of processes in motion including accessioning into the collection (dimension 3) and formalisation through documentation (dimension 2). This was immediately followed in 2011 by a realisation at Tate (dimension 1) resulting in further re-formalisation of the works characteristics negotiated with the artist (dimension 2) and changes to the underlying technical system—namely the need to constrain regularity of printing operations—to accommodate display in a busy gallery (dimensions 1). These technical alterations (dimension 1) were carried out in collaboration with the programmer who authored the original code (dimension 3). This programmer was based in Peru, and a remote access system was used to allow him direct access to the display computer at a Tate site in London (dimension 3). As a work primarily exhibited in Latin American countries prior to acquisition (dimension 4), being realised in the context of a European (and predominantly English-speaking) country (dimension 4) resulted in a level of recontextualisation through language changes and new documentation (dimension 2). This realisation operated using a different database (dimension 1) which captured words from English language Google search results, rather than Spanish (dimension 4).

In this instance a trajectory from the events of Fujimori presidency through to a gallery installation in London many years later can be established. The ways in which this trajectory—particularly the works acquisition by Tate—has resulted in a degree of compromise and reformalisation of the work's characteristics is made clear. It also implies that the work's reliance on the Google search engine data stems from an interest in serendipitous association in meaning rather than an interest in the Google search engine per se, and as a result this aspect of the work—which is problematic in terms of conservation—might be open to interpretation if the work's future realisation demanded it. Perhaps most importantly, the narrative highlights the connection with the history of Peru and its social memory, and the need for care in

recontextualising this conceptually important element of the work.

The approach outlined in this section serves to highlight one potential application of continuum theory—or what might be better labelled continuum thinking—to conservation problems and the documentation of the life a software-based artwork. The model's dimensions are useful prompts for exploring artworks technical histories even when considered in isolation, but it seems particularly useful as a way of identifying the kinds of event which trigger cascades of influence through the dimensions, such as those associated with the acquisition of *Brutalism*. This may be helpful in identifying when to revisit an artwork biography and remap these trajectories. Earlier observations regarding software evolution hint at the occurrence of a kind of punctuated equilibrium: periods of relative stasis are interspersed with periods of rapid change, with recurrent causes for such events. These examples provide further evidence that acquisition and display are among the most important of these events. It should also be acknowledged that my formalisation is just one potential view on the continuum among many possible. Much like the artwork biographies of van de Vall et al., understanding the continuum requires accepting the inherent non-neutrality of individual accounts and the "standpoint of the writer" (van de Vall, et al., 2011, p.7). Gathering multiple biographical perspectives will therefore serve to create a richer historical record—the artist's own biographical fragments being one perspective of clear interest.

6.6.2. Capturing Conservation Narratives in Practice

While emerging from a project involving conservation practitioners, the biographical approach developed by van de Vall et al. (2011) remains primarily theoretical and is not immediately reconcilable with the day-to-day of the conservator's professional role. In this section I will consider the practical implications of the principles of artwork biography and address the question of how they might mesh with conservation activities in practice. While new forms of art and media demand the reconsideration of many established processes, the museum conservator has been telling stories about the technical histories of artworks for some time—endeavours which are now widely understood as constituting the field of *technical art history*. Erma Hermans defines this as an area of study which:

> "aims at a thorough understanding of the physical object in terms of original intention, choice of materials and techniques, as well as the context in and for which the work was created, its meaning and its contemporary perception." (Hermens, et al., 2012, p.165)

Technical art history exists at an intersection of interests, so bringing together conservators, art historians and specialists from other fields—much as we might hope from a biographical approach. The origins of technical art history lie in so-called *technical studies* of artworks, which were often carried out as part of conservation work (for example the studies published in the National Gallery's Technical Bulletin series (anon. The National Gallery Technical Bulletin, 2017)). Indeed, in some cases, technical art history is defined directly in relation to the "scientific examination of works of art ... [by] researchers from the fields of art history, conservation, and conservation science" (Ainsworth, 2005, p.5). Where "scientific examination" in the context of traditional media might introduce interdisciplinarity through exchanges with chemistry (for painting) or geology (for sculpture), a reframing for software-based artworks might draw on many of the computer science related approaches discussed and developed in this thesis.

As a result of a historical association with conservation, much of what might be considered technical art history also fits within the range of activities expected within the discipline of conservation. Conservation, after all, requires close technical study of medium and methods. Due to this similarity, many techniques used in developing conservation documentation may also offer insight into technical art historical concerns. Recent research indicates that this may also apply to the conservation of software-based art. Deena Engel and Glenn Wharton have already demonstrated this kind of synergy elegantly, in a paper on technical art history revealed through conservation-driven source code analysis of software-based artworks at the Museum of Modern Art in New York (Engel, & Wharton, 2015). If, as these authors suggest, the nature of methods for the examination of technical art history overlaps with those of analysis and documentation within conservation, there seems to be solid grounds for extending conservators' activities to encompass production of art historical narratives that broadly align with the biographical approach explored earlier in this chapter.

Technical art history has a strong history at Tate through its research and conservation departments. Tate publishes public facing technical art historical information for selected artworks through what are referred to as "Technique and Condition" texts. These are available through the Tate website and collection catalogue, and are in essence brief technical accounts of a work's making and conservation history, written for a non-expert audience. Jo Crook, former Conservation Curator at Tate, introduces them in an internally published introduction

to the writing of these texts:

> “A technique and condition text is a summary of the making, technical structure and where relevant the condition of a work in the Tate collection, written for Tate online and accessible to a general non-technical audience and also of interest to specialists.” (Crook, 2015)

The structure of the reports is broken down into two sections: “materials and techniques” and “condition and treatments”. The materials and techniques section offers a narrative account of the elements that make up the work and how they were created, while the condition and treatments section presents a description of the condition and history of conservation treatments as far as is known. Taking the Tate Technique and Condition text format—which had yet to be explored for Tate’s software-based artworks (or any time-based media artworks)—as a basis, I have written texts for five of the artwork case studies in order to test its suitability (see Appendix III).

In constructing these texts, I found that in many cases new sources of documentation had to be considered. Indeed, there are an array of relevant materials existing on the edges of conservation practice which are required to support a technical art history of software-based art. The potential value of contextual materials in collecting and caring for time-based media artworks has been highlighted by a number of authors—curator Steve Dietz comments on the potential value in preserving “materials that might have been linked to the work” (Dietz, 2014) while conservator Ben Fino-Radin highlights the interest of “ephemera produced by the artist” (Fino-Radin, 2011, p.20). These might include production materials, the artist’s websites and other online activity or even, as Fino-Radin suggests, artist’s working computers. Another important component which has been little discussed in the context of preservation, is the relevance of the complex histories of third-party software and other technical components which might not be considered part of the artwork. For instance, programming languages are (much like the software they are used to produce) evolving, and the documentation of these language at any one moment represents a snapshot of the language’s specification in time. While this provides the *how*, understanding *why* it was used at a particular time will requires new forms of scholarship which engage with the history of software development.

While extending the supporting body of documentation represents a source for the development of narratives of technical art history, the formal structure of the

Technique and Condition text was found to be restrictive when attempting to convey their complexity. This is partly because these texts are short public-facing summaries, and the level of detail that it is possible (or indeed, desirable) to convey within them is limited. However, it also reflects fundamental challenges in constructing static narratives of technical history for artworks which have the potential to change in their makeup, and even ontology, during their life in the museum. Artwork biographies must necessarily vary in their structure, to accommodate the "different beginnings, itineraries, dynamics and endings" presented by complex artworks (van de Vall, et al., 2011, p.6). Representing the multiple versions, variants and realisations of a software-based artwork as they emerge through time requires reconsidering the form of that narratives of conservation and technical art history take. Approaches such as the use of the Wiki—a development in documentation management which paralleled the software version control system (Fuller, & Yuill, 2008)—have recently been explored by media conservator Martina Haidvogl and other collaborators at San Francisco Museum of Modern Art (SFMOMA) (Johnson, 2016). At SFMOMA it is being used as a tool to help manage the documentation of time-based media and other complex artworks. The dynamic, collaborative and flexible nature of the Wiki paradigm may make it similarly well suited to supporting the conservation narratives that conservators of software-based art may wish to capture and convey.

6.7. Chapter Summary

In this chapter I have developed a theoretical framework to guide the documentation of the patterns of evolution that occur in the life of a software-based artwork. Taken together, the contributions can be used to direct a documentation approach that charts the evolution of the work through time. In this first part of the chapter I introduced the idea of the metaphorical 'life' of the work through two conceptualisations: lifecycle and continuum. In practice, both are useful in providing insight on the patterns of change that can be observed within the lives of software-based artworks.

A lifecycle perspective helps us to understand that evolution often occurs in relation to certain life events, such as acquisition or display. Patterns of evolution vary between artworks and can be understood in relation to principles of software evolution (an area of study within software engineering), which suggest that highly specified software is less likely to evolve than that which is in some way reflexive of or embedded within human activity and external environment. Where evolution does occur, it can be understood as occurring on two inter-related levels. At the lowest

level, logical constructs—code, environments and interfaces—are manipulated to affect incremental change. These can be documented using systems-driven change tracking. Micro-level change patterns yield new identifiable variants of the software, which at the higher macro-level can be understood in relation to well known transformation types from the domains of software development and digital preservation. These can be documented using new vocabulary incorporated into existing frameworks for recording production history. This is aided by a clear conceptual model for structuring the relationship between artwork and its expressions and realisation, which I developed in Section 6.4, based on a model from descriptive bibliography.

Despite clear uses for the largely systems-driven approaches discussed in the first part of this chapter, the case study artworks examined reveal that change can only be truly understood in relation to the rich socio-technical context of software-based artworks. Building on ideas from artwork biography and continuum theory, I developed an approach to capturing narratives of technical history which engages with the various external forces that shape the ongoing processes of creation and formalisation in the evolution of the artwork. In practice, this information may reside in multifarious forms, and be supported by an array of contextual materials which may not conventionally be sought out by those caring for software-based artworks. Approaches to managing complex, multi-authored documentation, such as the Wiki are beginning to find favour in museum environments, and show promise in helping to deal with the issues of connectivity and change management that limit the capture of narratives of software-based artwork evolution. Existing modes of conservation storytelling might also be reframed, as demonstrated in my explorations of the Technique and Condition text methodology used at Tate. This requires renegotiating traditional museum models in order to accommodate the levels of change that are occur for software-based artworks (and other forms of time-based media), and make clear the role of the conservator in shaping the life of the work.

CHAPTER 7

CONCLUSIONS AND RECOMMENDATIONS

7.1. Research Contributions and Applicability of Outcomes

Based on the identification of a gap in existing scholarship, the aim of this thesis has been to develop approaches to the documentation of software-based art in support of its conservation and long-term preservation. Through practice-led research—grounded in a set of case study artworks from the Tate collection—and the synthesis of theory from several related domains, the outcomes of this research are intended as contributions to theory and practice in the fields of art conservation and digital preservation. In this section I will reflect on these outcomes and their respective research contributions and theoretical connections, while considering the extent to which these outcomes may have wider applicability beyond this research. This forms the final component of the constructive research methodology adopted in this research (corresponding to Stages 4 and 5).

In Chapter 2 and 3 of the thesis I developed a conceptual framework for understanding the two foundational elements of this research: software as a material and medium; and the nature of the document in the context of conservation. This

conceptual framework allowed the development of a more nuanced understanding of the problem space identified in Chapter 1, and guided the shape of the chapters that followed. Chapters 2 and 3 also contain research contributions that stand alone, and I reflect briefly on these below before discussing the primary research outcomes of Chapter 4, 5 and 6 in the subsequent sections.

In Chapter 2 I brought together existing knowledge and theory relating to software as a technology (drawing particularly on the computer science domain) and as an artistic medium (drawing particularly on art conservation theory and the history of media art). In doing so I developed a comprehensive understanding of the considerations and challenges the medium presents to conservators faced with its long-term care. I proposed that software might be best understood as possessing multiple material statuses, all of which are of concern to conservators. At the lowest level it is a physical representation of bits, stored using a physical carrier. At the level above, it is a symbolic construct—code—which can be considered as analogous to a score or script. Higher still, it can be understood as a software process: the execution of the code within a suitable technical environment, which yields the experiential elements of the artwork's software component. These experiential elements are the highest level at which we can understand software, and can be termed a software performance, itself a part of the artworks larger realisation.

Formalised as the software performance model, the processual perspective taken is one way of understanding the link between an artwork's concrete elements (such as hardware components or software binaries) and the ephemeral nature of the experience of the work when it is realised in time and space. The intangible and contingent processes which produce this performance introduce potential variability into each realisation of a software-based artwork, through the effects of a variable constellation of hardware and software components. One way of seeing the goal of the conservator then, is to maintain consistent software performances through time, despite changes in the other components of the model (such as code or technical environment). This novel outlook is also relevant to other kinds of performative computational phenomena, such as video games and commercial software products. The lexicon developed in the last part of the chapter is the other primary contribution of this chapter (p.63). This describes a set of terms which I identify as the primary conservation considerations posed by software as a medium, and to which strategies for documentation must respond. This extends current understanding of the medium within the field of conservation and may also be of interest to those who work closely

with software-based art in fields outside of conservation such as art history and media theory.

In Chapter 3 I focused on defining potential forms of documentation and understanding their significance within conservation—so developing the second part of the conceptual framework. In the first part of this chapter I considered documentation theory from the 1950s to the present and looked at how the scope of the document concept might be pragmatically defined for museum conservation documentation, particularly where this might be challenged by the characteristics of software-based art. I found that the inclusive definitions developed by early pioneers of what is now information science are still very much relevant to the way we understand documentation today, particularly in their positing of documents as primarily defined by *use*. Much of this chapter was dedicated to defining the problem space that the remainder of the thesis sought to address, and so its relevance is rather specific to the context of this thesis. However, this examination of documentation theory represents a small contribution to the current resurgence of interest in this area and may of particular interest to those working at the intersection of information science and cultural heritage.

Drawing on the structure of institutional conservation workflows, in the latter half of the chapter I sought to explore the types of documentation used within the conservation domain and appraise the suitability of existing documentation models for describing software-based art. While this part of the chapter was, again, primarily a grounding component of this research, it also represents a contribution to our understanding of time-based media conservation practice today, within which the position of document has not yet been extensively recorded or studied.

Within Chapter 3, I identified that, while frameworks exist to support general documentation activities within the conservation of time-based media art, there has been little attention given to the specific considerations presented by software as a medium. Through this analysis I identified three broad documentation challenges which formed the focus of the next three chapters:

- Software is structurally complex and closely linked to the technical environment in which it is executed, and understanding and documenting these structures is crucial to the preservation of software-based artworks. How can this information be effectively derived and represented?

- Changes to some of the components of a software-based artwork are expected to occur in their long-term preservation. How can documentation be used to ensure that the core identity of the work is captured and appropriately managed through time as it is realised in different contexts?

- Software-based artworks are the result of processes largely unfamiliar to collecting institutions, and the works themselves are likely to continue to evolve through time while within their care. How can the evolution of the artwork through time be captured by conservators as documentation?

These interconnected focal areas correspond to three broadly defined outcomes of this research, which I derived in Chapter 4, 5 and 6. In the following three sections I consider the value of these outcomes in more detail, identifying the main theoretical and practical contributions as well as their potential limitations.

7.1.1. Binary-centric Analysis and the Software-based Art Structure Ontology

The technical structures which underpin any one software performance are complex systems consisting of numerous software and hardware components in a particular configuration. The software binaries which are executed within a performance appear essentially opaque, in that the instructions encoded within them cannot be readily interpreted by a human, and so the details of their functionality and connectivity are concealed. Furthermore, the software performance exists only as the result of an ephemeral process, as code instructions are executed by the host computer system in real time. Source code analysis has been established as the primary means of decoding the functionality of software-based artworks and has been demonstrated to be a powerful tool in prior research. I do not dispute the value of source code analysis as a process in software documentation—it is evidenced by a long history of use in software development where it is highly valued by software engineers and is now further supported by practice-led research in the field of art conservation. However, in Chapter 5 I explored a number of situations in which utilising this kind of approach might be challenging and developed a three-part critique which indicates a requirement for other approaches to supporting analysis and documentation processes (p.107).

In response to this critique, I explored a set of methods which offer a counterpoint to source-centric analysis and focus instead on the environment-embedded executable representation, bypassing the need for access to source code (from p.116). In these

cases, methods from software development and reverse engineering may be repurposed to step into the software as it is executed—logging events and tracing program flow—and even reverse the compilation process. Careful analysis of data generated can yield important insights for conservation, including elucidating complex dependencies, revealing unclear program behaviours and capturing significant performance characteristics and metrics. This approach could have particular value for conservators who utilise environment-centric approaches to preservation such as emulation and virtualisation, where the focus is on understanding and reconstructing environment rather than migrating the code, which would require a deeper understanding of functionality. While presented as an alternative to source code analysis, it is important to acknowledge that there is no single universally effective method for analysing software. Rather, there are an array of complementary tools which, when combined with an informed human interpreter, are more than the sum of their parts. The approaches introduced here provide another set of tools, and are likely to be of considerable interest to conservators of software-based art.

Decision-making regarding which tools to use in a particular scenario may not be straightforward however and relates not only to the questions that must be answered, but the expertise and resources available. In this respect, the effective use of source-centric analysis approaches is contingent on the availability of expertise in the particular language and type of use. The effective use of binary-centric approaches on the other hand, often comes down to problems of data volume and identifying the pertinent information within that data. Getting concrete answers may require highly specialised knowledge of assembly language and the associated time investment required to carry out reverse engineering at this low level. The potential for a generalist software-based art conservator is therefore unclear. Anyone with an understanding of one high-level programming language (in which all the case studies I examined were programmed) is likely to be able to read a program written in another, given sufficient time to learn its constructs and syntax. However, languages are many, and time and resources are not unlimited, meaning that museums will be required to find a balance between the development of expertise within their conservation departments and the fostering of new collaborations outside the institution.

Another limitation to binary-centric approaches is that they involve interpretation or translation by third party tools, as opposed to source code which has a more direct link to the process of creation. This introduces a certain amount of uncertainty about whether the information gained is accurate or complete. As such, selecting

instrumentation techniques for a particular artwork requires careful assessment. This is all exacerbated by the fact that this kind of software analysis exists on the fringes of mainstream computer science, as it can be applied to the illegal reverse engineering of proprietary software projects developed in the commercial sector. Indeed, whether such methods are usable at all will often be dependent on whether community developed tools are available for the specific purpose and platform in question. Conservators of software-based art are, at the current moment, fortunate in that the artists are usually still alive, and are willing to engage with them directly to help preserve their work. There is no reason that this kind of collaboration should not continue to be an important part of the conservator's role. In some cases, then, it may be possible to consider the analysis of process and the principles of instrumentation in collaboration with the artist. Developing and defining suitable analysis techniques in these collaborations may be the best route for ensuring that future realisations of the work maintain the appropriate functional and non-functional requirements.

In the last part of the chapter, I developed a conceptual model for capturing component-level metadata representations of software structures using information derived from software analysis (p.139). This model provides a semantically meaningful formal language which might be incorporated into information systems such as collections management systems or digital repositories. The use of this model is an advancement of earlier efforts, not only in that it incorporates formal semantics, but because it is designed for a software preservation use case which has so far received limited attention. In practice, it is likely to be of particular interest to conservators as a tool for recording structured information to help locate and identify the hardware and software components of software-based artworks, and in supporting the reconstruction of appropriate technical environments when applying emulation and virtualisation strategies. This is of relevance to the preservation of other kinds of software system beyond the realm of art conservation, where such use cases are also poorly served by existing standards.

The most apparent limitation of the model is that, in the current museum climate, museum information systems may not be sufficiently technically developed to incorporate ontologies, making it challenging to integrate with existing systems. However, it seems likely that this may change in the near future as information systems continue to develop, given a growing interest in such approaches within museums. Elements of the ontology may also inform the structuring of simple (not ontology based) metadata schema, vocabularies and thesauri, particularly the set of

component classes and their hierarchical structure. Furthermore, the ontology stands alone as a theoretical model which captures the structural form of software systems, and so furthers discussion regarding the most appropriate way to do this in the field of digital preservation.

7.1.2. Significant Knowledge and the Requirements Specification

The management of change is understood to be an essential consideration in the conservation of software-based art and must be supported by appropriate documentation— documentation which captures something of the artwork's identity, so that it can be maintained through time as change is negotiated at a technical level. In Chapter 4, I developed an approach to the capture and management of the identity of a software-based artwork through time using documentation. This is not a new problem in art conservation or digital preservation, and the *significant properties* concept provided a starting point for this discussion. In reviewing existing literature, I found that while there are problems applying this approach in practice due to the subjectivity inherent in property definition and unclear guidelines for implementation, the fundamentals of the concept might be usefully reframed as *significant knowledge* (p.155). This unloads some of the historical baggage associated with significant properties and broadens the concept's scope to encompass the diverse array of documentation and other (potentially tacit) knowledge sources which support the long-term care of an artwork.

Using a set of significant knowledge categories developed in this chapter as a guide, artwork and medium specific approaches might be applied as appropriate, ranging from the acknowledgement of the tacit knowledge present in individuals within an organisation, to the definition of metrics for verifying software performances at a technical level. This approach better reflects the reality of conservation practice as inherently subjective, necessarily bespoke and responsive to emergent forms of software-based art. It is important to note however, that this does not make attempting to capture documentation that represents an artwork's identity trivial, and the inherent challenges to formalising such a concept remain the primary limitation in developing documentation of this kind. The knowledge categories proposed nonetheless represent a small but significant shift in thinking, which responds to earlier criticisms of significant properties (and associated theoretical stasis) and may be of particular interest to the digital preservation community as a means of working with significant knowledge in practice.

While many of the significant knowledge categories identified were well supported by relatively well understood documentation types, those categories relating to the software performance itself were found to lack a clearly defined means of documenting how software should behave. To meet this need, I proposed a reframing of the *requirements specification*, a loosely defined documentation artefact widely incorporated into software development processes (p.161). This approach aims to specify what a software system needs to do (or its functionality) as *functional requirements*, and the constraints on how it does them as *non-functional requirements*. By avoiding reference to specific technologies except where this is necessary, a requirements specification allows developers to choose a technical platform that is appropriate for implementing the requirements. In a conservation context, we consider the place of requirements not as something specified prior to development and in support of software, but rather as a tool for supporting processes analogous to software system maintenance. The requirements document can be used to provide a clear and maintainable record of what a software program is meant to do when realised, and within what constraints, making clear the permitted parameters for flexibility and change across realisations.

The exploration of requirements as a documentation tool in this thesis represents the first detailed work to consider how this component of software engineering might be utilised within an art conservation context. In some cases, the value of a requirements engineering approach may be limited, particularly where a work is very closely linked to a particular technology and so is very difficult (or even impossible) to separate from it without compromising the identity of the work. Put another way: the more closely linked a software-based artwork's identity is to its actual implementation by the artist as software, the harder it would be to migrate it to another technology using functional requirements as a basis. In these cases, however, the specification of non-functional requirements can still be valuable in ensuring that the work is appropriately realised and that the characteristics of the software performance are maintained through time, when changes in its technical environment occur (for example, if it is emulated). The transformation of software into a set of requirements can also be a valuable investigatory tool, as this formalisation, when combined with rigorous examination and artist consultation, can help clarify the relative significance of the characteristics of a software performance and their relationship with the identity of the work.

Much like significant properties, the utility of requirements in practice is challenged by the need to exhaustively identify them, or risk compromising the identity of the work—

should, for example, the requirements be used as the basis for a conservation treatment. When the artwork is in its latent state between realisations—usually as a set of components in storage—it may be particularly difficult to identify requirements, particularly for installed works. Although to some extent mitigatable through collaboration and transformation of tacit to explicit knowledge, it is also important to acknowledge the degree to which such an approach is subjective. This is a particularly important limitation to note, as at the point of requirements specification the conservator must make decisions regarding the target layers for preservation, particularly the weighting of preserving the technology against preserving the performance. Requirements might therefore be best captured when a work is displayed and with artist involvement and approval. The latter may be challenging however, as the language of requirements engineering is not likely to be that of artists who typically work outside the structures of formal software engineering. Nonetheless, the fundamental approach of separating functional and non-functional requirements appears to have a resonance with the documentation demands of software-based art conservation. With its long history and continued place in software engineering, the concepts at the heart of requirements specification offer a useful theoretical framework for the conservator. Indeed, its principles have the potential to be further extended to the description of other time-based media artworks where technology takes on a primarily functional role.

7.1.3. Change Models and the Sociotechnical Biography

Software-based artworks, much like other forms of time-based media art, are contingent on a process of realisation for them to be experienced and can only be truly understood as unfolding through time. This unfolding occurs not just in the performance of their media components (e.g. the execution of a software program) at the time of realisation, but in the evolution of the artwork itself as it is realised at different points in time. Software creates an additional level at which change might occur between realisations, and to maintain a documentary record of this evolution, it is important to have a clear understanding of the nature of these processes and their relationship with the artwork. In Chapter 6, I explored how software-based artworks evolve through both iterative and transformative processes of change, and how this evolution might be captured as documentation. I situated this discussion in relation to several theoretical perspectives on change, including contrasting lifecycle and continuum models, and theories of software evolution.

I found that established terminology and methodologies from software engineering

can be directly applied to documenting technical processes of change for software-based artworks. Version control systems (VCS) used in software development processes offer a particularly powerful tool, and the data captured by these systems can represent a record of a conservation intervention, and a rich resource for technical art history. While this component of the research was primarily of an exploratory nature, it is clear from preliminary evidence that this approach allows capture of authorial, temporal and descriptive information which would otherwise be lost. The value of these methods in conservation practice may be limited by how well they mesh with the working practice of an artist—where this kind of direct collaboration continues during the works life in the museum—or with that of programmer collaborators. They may also be less useful for documenting development processes which are not code based, as while VCS systems might recognise that a non-text file has changed, the nature of the change may be lost unless this is recorded manually. At a higher-level, processes of software development might be understood in relation to transformations in the software itself. I introduced a set of terms for classifying changes based on language from software engineering, which might be used to record transformations occurring in the history of a software-based artwork. These terms are likely to be useful to conservators writing documentation which records media production histories for software-based artworks.

Moving to a higher-level still, I looked at how the relationship between the software-based artwork and its multiple forms might be captured. Using mature models from bibliographic records as a basis, I developed a conceptual model for describing the hierarchical relationships between the work and its various expressions (p.204). This model also extends the ontology developed in Chapter 4, and provides formal language for the linking of the software structures described in that chapter to related realisations, variants and versions. This model is a theoretical contribution to ontology in the conservation of time-based media art, while also being of interest to conservators and information professionals considering the integration of version information with collection-related information systems. While the model is intended to be generic enough to describe the considerable diversity of form found in software-based art, including the case studies examined, only its continued use in practice will allow more concrete conclusions regarding its utility. The primary limitation of this model and to some extent the others discussed above, is that they suffer from limitations regarding the extent to which they can capture the human and social context in which software evolution occurs. Without this context, the understanding of

the evolutionary history of an artwork that can be gained from documentation is partial.

To address this shortcoming, I proposed that a more contextually rich, narrative-driven form of documentation might be employed. Drawing on the notion of the artwork biography from conservation theory, and applying principles from records continuum theory, I demonstrated an approach which focuses on the creation of biographical fragments (p.211). These fragments draw links between the artwork, including its technical components and characteristics, and the various forces that shape its evolution through time. This represents a demonstration of how artwork biography and continuum theory, which have primarily existed only as theoretical frameworks, into practice in the description of software-based artwork life histories. The primary limitation of this approach is that it is more resource intensive than other approaches discussed in the chapter and does not readily integrate with existing notions of day-to-day conservation practice. While I propose various ways in which this might be aided and incentivised (such as the desire of museums to create public-facing narratives of technical history), ultimately the in-depth research required to generate these kinds of biographies remains time intensive and highly specialised. Nonetheless, their contribution to the documentary record of the work is likely to be unique, and their creation may form an important part of the conservator's role in illuminating the history of the work and its treatment. This approach, and the example narratives generated, are a contribution to the emerging field of technical art history, and may be of interest to those working in areas of scholarship where the technical history of software is also studied and reconstructed, such as software studies and software archaeology.

7.2. Reflections on Overarching Themes

In Chapter 2, I suggested that in understanding the inherent performativity of software as a medium, we might frame the role of the conservator as working to ensure that, for any one artwork, a consistent software performance can be achieved through time. Through this lens, we can consider how elements of the software performance model that result in the performance—the source code, the technical environment, and the computational process—might be permitted to change, providing the identity of the work that resides in the software performance is maintained. Looking at this research as a whole, we can see that by understanding the connection between the way software has been used by the artist and the characteristics of the software performance, different kinds of documentation come to the fore.

A work might emphasise the consistent realisation of a precisely defined core identity each time it is realised. In these cases, it may be most appropriate to maintain the software super-object as-is, and by carefully analysing and documenting this objects relationship with its technical environment, ensure that these critical links are maintained. In other cases, the artwork may employ software in a way which is functional (that is, designed to carry out some task or implement a particular algorithm), and so permit the reimplementation of the functionality represented by the source code using another language or tool. In other cases, a work's identity may be so closely tied to its socio-technical environment, that it must evolve in order to stay alive. In these cases, it may be accepted that the work will live on outside the collection, so shifting the emphasis of conservation work to capturing documentation that represents its historical states. In practice, the weighting of these different concerns may shift over time, so requiring a reconsideration of the focus of documentation efforts. Regardless of the nature of the changes in the artwork through time, documentation is an important legacy for institutions collecting and conserving software-based art. In a sense, the generation and collation of documentation could be considered a preservation strategy in and of itself, which focuses on capturing a work's complex sociotechnical history, and a representation of the work as record or trace.

Variety among software-based artworks, as described above, as well as the cultures of the institutions which collect them, will result in a proliferation of approaches needed to care for them. Developing any single comprehensive strategy for their long-term care is of course, impossible, and this research represents a set of contributions to a challenge which cannot be solved, but rather must be regularly and collectively reconsidered. In the conservation of time-based media art, it seems that specific formal approaches to documentation are rarely adopted universally. In an interview, Jon Ippolito summed up his feelings on the legacy of his Variable Media Questionnaire:

> "People show me their questionnaire, and at a certain point I realise, 'You know, isn't that the point?' That people start doing it. They don't have to do it my way, as long as they do it their way." (J. Ippolito, personal communication, 9 February 2017)

Here Ippolito acknowledges that while it is tempting to focus on the value and adoption of a specific approach, it tends to be the overarching theoretical outlooks that have wider influence. In a way, the Variable Media Questionnaire has become a kind of design pattern for describing media art, one which focuses on twin principles

of medium-independence and the artist interview.

With this outlook in mind, the value of this thesis is not in providing a rigid model or a set of document templates; instead its contents might loosely be considered as a set of design patterns. The idea of pattern has its roots in physical architectural and human-oriented design, and stems from the research of Christopher Alexander into town and city building (Alexander, 1977):

> “Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.” (Alexander, 1977, p.x)

The intention of a pattern then, is not to provide a rigid model or template. Rather, the pattern is a modular, flexible and semi-user-defined solution. It is my hope that the research outcomes highlighted in this chapter might serve as patterns for the formulation of case-appropriate conservation documentation for software-based art.

At the very least, this thesis has served to fill a knowledge gap in understanding the kind of documentation a conservator might expect or hope to receive when a software-based artwork falls into their care. Better still, this same information might help them recognise what is missing and provide a guide as to how it might be generated. This is important, as conservators will not always have access to the required level of documentation to enable them to carry out their professional role. In this sense, this thesis also provides a reference work for the state of software-based art conservation at a point in time. As collaborations and knowledge exchange with software experts builds a shared understanding of the issues at play, we will hopefully see the development of even more refined—and potentially community driven—frameworks within this domain.

The time-based media art conservator has always had a hybrid role, balancing technical concerns with respect for artistic intent and the practicalities of *just keeping things running*. The nature of software-based art conservation involves developing more facets to this role, which might move fluidly between software archaeologist and performance dramaturge. A defining element of this role is collaboration, and handling software will undoubtedly require new links to be forged in the both commercial and public sectors. There is also the need for new technical skills from computer science and software engineering in the field, although the precise way in which these will manifest—i.e. what hybrid of collaboration, training and consultation—remains

unclear. My own research indicates that, in the short term at least, the role may require the development of new skills which resemble those of the systems administrator, such as the configuration and maintenance of software systems. While many of the lessons learned here are broadly applicable to software in all its forms and uses (which have certain shared technological foundations), this research responds to a particular moment in time. If the museum is to keep pace with the rapid emergence and adoption of new technology by artists, which historical evidence indicates often outpaces museums acquisition strategies, another facet to their role will be a continued engagement in the cultures and communities of software development.

7.3. Recommendations for Further Research

As a thesis intended to generate outcomes with practical implications, I want to conclude by offering a set of recommendations for future research. The first is an acknowledgement that the documentation patterns identified in this research would benefit from further testing order to make the final jump from theory to practice. This would be best carried out through consultation with conservation practitioners, perhaps through focus groups or independent review, and should focus on assessing the extent to which they might be operationalised. The other recommendations I will make pertain to specific avenues of research that extend the work started here, and that I feel would offer a significant contribution to the still emerging field of software-based art conservation. They would also contribute to a broader knowledge base from which to better understand the difficult problem of software preservation, as applied to a range of software types in addition to software-based art.

Recommendation 1: Explore the feasibility of shared infrastructure for generic components of software environments.

Essential in the future of software preservation, is access to the software of the past. The software structures described in Chapter 5 are carefully constructed environments, contingent on software components which may be highly specific. Many of these are commonly reused across environments however, and are liable to be repeatedly drawn on for future preservation efforts (particularly those employing emulation and virtualisation) by different institutions and in different areas of digital preservation. For example, there are sets of common operating system families, including numerous versioned products, which form the basis of most software structures. Particular dependency sets are also recurrent, such as runtime libraries,

runtime environments and hardware drivers. Similarly, maintaining availability of popular development environments and production software would help ensure long-term access to source projects. The availability of a shared repository containing reusable and well described copies of these software components could be a valuable resource for the digital preservation community, avoiding the sometimes arduous process of locating legacy components online on through resale. Unfortunately, the creation of such a library in any centralised and openly accessible form is likely to be severely limited by legal constraints on the redistribution of software—obstacles which will require negotiation with the original producers in order to be solved. More immediately, it is important that those collecting software begin assembling their own supporting software libraries (legally) to ensure that these important artefacts are not lost.

Recommendation 2: Test the migration and rewriting of software-based artworks with complex functional requirements.

As I have shown within this thesis, not all software-based artworks are closely tied to a specific implementation of the software employed. They can instead be understood in relation to functional requirements, which the software implements as a means of achieving particular set of behaviours or characteristics. While this implies the potential for a degree of acceptable change at the software level, there are few practical examples of actually migrating and rewriting software with complex requirements from within the art conservation field. As a result, the extent to which this is possible without compromising an artwork's identity is not well understood. Undertaking practical experiments in these processes using real-world case studies is resource intensive work, but the insights gained from such research could be valuable in developing the conservation discipline's understanding of these issues. It could be particularly interesting to experiment with how documents such as the requirements specification could be used as a tool in the process. Testing could also engage directly with the artist and assess how requirements specification might act as a way of formalising the artwork's identity between realisations. More generally, it may be useful to experiment with the integration of principles of requirements specification (including use case description) into conservation documentation processes.

Recommendation 3. Further in-depth research into the technical history of software-based artworks and the way in which these narratives might be conveyed to various audiences.

This research has intersected with concerns of technical art history—an area of scholarship closely linked to conservation practice—in a number of ways. In general however, this remains an under-developed area of research in relation to software-based art. This is concerning given the ephemerality of many of the materials which offer the insights required to document such histories. Understanding production history involves gaining access to project files, production assets and prototypes, which often rely on particular technical environments for access, or on direct engagement with the artists working practice. Other ephemeral resources of relevant information, such as artists' websites and third-party software documentation, are in a constant state of flux and older versions are not necessarily archived by their maintainers. There is fertile ground for new strands of research here, and I recommend further in-depth research into the technical history of software-based artworks by conservators engaged in their care. Generating narratives of technical art history is an activity that has been closely connected with the role of the conservator historically. In order to develop this facet of conservation for software-based art, there is the need to allow conservators the resources to develop and pursue this important aspect of practice. In achieving this, it may also be necessary to develop approaches for conveying narratives of conservation and technical art history to general audiences. This may demand new models of documentation which move beyond static texts, and into dynamic forms of document such as the Wiki. Understanding how conservation knowledge might be made public through a Wiki or similar system of knowledge management could be another fruitful area for future conservation research and collaboration.

BIBLIOGRAPHY

Act-3D (2012) *Quest3D Front Page* [online]. Available from: https://web.archive.org/web/20170822144850/http://www.quest3d.com/ (Accessed 30 January 2018).

Adang, L. (2013) *Untitled Project: A Cross Disciplinary Investigation of JODI's Untitled Game*. [online]. Available from: http://media.rhizome.org/artbase/documents/Untitled-Project:-A-Cross-Disciplinary-Investigation-of-JODI%E2%80%99s-Untitled-Game.pdf.

Adobe (2017) *Flash & The Future of Interactive Content* [online]. Available from: https://blogs.adobe.com/conversations/2017/07/adobe-flash-update.html (Accessed 7 September 2017).

Ainsworth, M. W. (2005) From Connoisseurship to Technical Art History: The Evolution of the Interdisciplinary Study of Art. *The Getty Conservation Institute Newsletter*. 20 (1), 4.

Alderson, A. & Shah, H. (1999) Viewpoints on legacy systems. *Communications of the ACM*. 42 (3), 115–116.

Alexander, C. (1977) *A Pattern Language: Towns, Buildings, Construction*. New York: Oxford University Press.

Alexander, I. F. (1997) A Historical Perspective on Requirements Engineering. *Requirenautics Quarterly: The Newsletter of the Requirements Engineering*. 12 (3), 13–21.

American Institute for Conservation of Historic and Artistic Works (1994) *Code of Ethics and Guidelines for Practice*. [online]. Available from: http://www.conservation-us.org/docs/default-source/governance/code-of-ethics-and-guidelines-for-practice.pdf?sfvrsn=9 (Accessed 11 February 2018).

American Institute for Conservation of Historic and Artistic Works (2016) *Conservation Terminology* [online]. Available from: http://www.conservation-us.org/about-conservation/definitions#.WU7cSWjyuUk (Accessed 24 June 2017).

Anon (n.d.) *About - Digital Preservation (Library of Congress)* [online]. Available from: http://www.digitalpreservation.gov/about/ (Accessed 23 January 2018).

Anon (2016) *Apache Taverna* [online]. Available from: https://taverna.incubator.apache.org/ (Accessed 5 September 2017).

Anon (n.d.) *Conservation – time-based media* [online]. Available from: http://www.tate.org.uk/about/our-work/conservation/time-based-media (Accessed 29 July 2017).

Anon (2006) *ISO/IEC 14764:2006(E) IEEE Std 14764-2006: Software Engineering — Software Life Cycle Processes — Maintenance*. [Online] 1–46.

Anon (2011) *ISO/IEC/IEEE 29148:2011: Systems and software engineering — Life cycle processes — Requirements engineering*. [Online] 1–94.

Anon (2008) *ISO/IEC/IEEE Std 12207-2008: Standard for Systems and Software Engineering - Software Life Cycle Processes*. [Online] c1-138.

Anon (2011) ISO/IEC/IEEE Systems and software engineering – Architecture description. [Online] 1–46.

Anon (2018) *openFrameworks* [online]. Available from: http://openframeworks.cc/ (Accessed 1 March 2018).

Anon (2018) representation, n.1. OED Online [online]. Available from: http://www.oed.com/view/Entry/162997 (Accessed 23 January 2018).

Anon (2014) *The InterPARES 2 Project Dictionary* [online]. Available from: https://web.archive.org/web/20141002211915fw_/http://www.interpares.org/ip2/display_file.cfm?doc=ip2_dictionary.pdf&CFID=5710346&CFTOKEN=69123980 (Accessed 3 September 2017).

Anon (2017) *The National Gallery Technical Bulletin* [online]. Available from: https://www.nationalgallery.org.uk/paintings/research/technical-bulletin (Accessed 11 March 2018).

Anon (2012) *The Preservation of Complex Objects Volume 2. Software Art*. [online]. Available from: http://www.pocos.org/books/pocos_vol_2.pdf (Accessed 3 October 2014).

Anon (2014) *Web technology for developers - SVG attributes: shape-rendering* [online]. Available from: https://developer.mozilla.org/en-US/docs/Web/SVG/Attribute/shape-rendering (Accessed 3 August 2017).

Apple (2016a) *FileMerge*.

Apple (2016b) *Xcode*.

Arcangel, C., 2006. *Colors*. In collection of Tate, London (L02995)

Arcangel, C. (2009) *Colors Personal Edition* [online]. Available from: http://colors-personal-edition.coryarcangel.com/ (Accessed 2 August 2018).

Arcangel, C. (2017a) *Colors-Personal-Edition: OSX App to play a movie one horizontal line of pixels at a time*. [online]. Available from: https://github.com/coryarcangel/Colors-Personal-Edition (Accessed 1 March 2018).

Arcangel, C. (2012, March 14). *Re: Archving [sic] your work Colors*. [Email to Iolanda Ratti]. Copy in Conservation Folder for artwork L02995. Tate, London.

Arcangel, C. (2017b) *The Source Digest*. Arcangel Surfware.

Arcangel, C. (2013) *Things I Made: Code* [online]. Available from: http://coryarcangel.com/things-i-made/category/code/ (Accessed 1 March 2018).

Atkins, R. D. (2009) Copyright, contract and the protection of computer programs. *International Review of Law, Computers & Technology*. [Online] 23 (1–2), 143–152.

Badger, C. (2008). *Subtitled Public Code Description.* Copy in Conservation Folder for artwork T12565. Tate, London.

Bawden, D. & Robinson, L. (2012) *Introduction to Information Science*. London: Facet Publishing.

Beerkens, L., t Hoen, P., Hummelen, Ij., van Saaze, V., Scholte, T., Stigter, S., (2012) *The Artist Interview: For Conservation and Preservation of Contemporary Art. Guidelines & Practice*. Heyningen: Jap Sam Books.

Behrens, B. C. & Levary, R. R. (1998) Practical legal aspects of software reverse engineering. *Communications of the ACM*. 41 (2), 27–29.

Bergeron, J., Debbabi, M., Erhioui, M.M., Ktari, B., (1999) 'Static analysis of binary code to isolate malicious behaviors', in *Enabling Technologies: Infrastructure for Collaborative Enterprises, 1999.(WET ICE'99) Proceedings. IEEE 8th International Workshops on*. 1999 IEEE. pp. 184–189.

Birnbaum, D. & Arcangel, C. (2009) Do It 2. Artforum p.191–199.

Borges, J. L. (1999) 'On Exactitude in Science', in *Collected Fictions*. New York: Penguin Books.

Boutard, G. & Guastavino, C. (2012) Archiving electroacoustic and mixed music: Significant knowledge involved in the creative process of works with spatialisation. *Journal of Documentation*. [Online] 68 (6), 749–771.

Briet, S. (2006) *What is Documentation?* English Translation. Scarecrow Press.

Brown, P., Gere, C., Lambert, N., Mason, C., (2008) *White Heat Cold Logic: Early British Computer Art 1960-1980*. The MIT Press. [online]. Available from: http://eprints.lancs.ac.uk/id/eprint/55521.

Bryant, A. & Charmaz, K. (2007) *The SAGE Handbook of Grounded Theory*. London, UNITED KINGDOM: SAGE Publications. [online]. Available from: http://ebookcentral.proquest.com/lib/kcl/detail.action?docID=1138448.

Buckland, M. K. (1997) What Is a 'Document'? *JASIS*. 48 (9), 804–809.

Butterfield, A. & Ngondi, G. E. (eds.) (2016) software. *A Dictionary of Computer Science* [online]. Available from: http://dx.doi.org/10.1093/acref/9780199688975.001.0001 (Accessed 30 July 2018).

Cao, L. & Ramesh, B. (2008) Agile requirements engineering practices: An empirical study. *IEEE software*. 25 (1), . [online]. Available from: http://ieeexplore.ieee.org/abstract/document/4420071/.

Castriota, B. (2017) *Ontological Models and Authenticity in Time-Based Media Art Conservation*. [online]. Available from: http://www.academia.edu/32430089/Ontological_Models_and_Authenticity_in_Time-Based_Media_Art_Conservation (Accessed 28 January 2018).

CCSDS (2012) *Reference Model for an Open Archival Information System (OAIS): Magenta Book*.

Cerpa, N. & Verner, J. M. (2009) Why did your project fail? *Communications of the ACM*. [Online] 52 (12), 130.

Ceruzzi, P. E. (2003) Google-Books-ID: x1YESXanrgQC. *A History of Modern Computing*. 2nd Edition. MIT Press.

Chan, J.-T. & Yang, W. (2004) Advanced obfuscation techniques for Java bytecode. *Journal of Systems and Software*. 71 (1–2), 1–10.

Chen, S.-S. (2001) The paradox of digital preservation. *Computer.* [Online] 34 (3), 24–28.

Chikofsky, E. J. & Cross, J. H. (1990) Reverse engineering and design recovery: a taxonomy. *IEEE Software*. [Online] 7 (1), 13–17.

Chung, L., Nixon, B.A., Yu, E., Mylopoulos, J., (2000) Google-Books-ID: MNrcBwAAQBAJ. *Non-Functional Requirements in Software Engineering*. Springer Science & Business Media.

Chung, L. & do Prado Leite, J. C. S. (2009) 'On non-functional requirements in software engineering', in *Conceptual modeling: Foundations and applications*. Springer. pp. 363–379. [online]. Available from: https://link.springer.com/chapter/10.1007/978-3-642-02463-4_19.

Ciula, A. & Eide, Ø. (2014) 'Reflections on cultural heritage and digital humanities: modelling in practice and theory', in *Proceedings of the first international conference on digital access to textual cultural heritage*. 2014 ACM. pp. 35–41.

Clark, R., Frieling, R., Haidvogl, M., Scher, J., (2015) *Predictive Engineering by Julia Scher: A Case Study from the Artist Initiative, San Francisco Museum of Modern Art*. [online]. Available from: http://www.tate.org.uk/context-comment/video/media-transition (Accessed 12 March 2018).

Cook, S., Harrison, R., Lehman, M.M., Wernick, P., (2006) Evolution in software systems: foundations of the SPE classification scheme. *Journal of Software: Evolution and Process*. 18 (1), 1–35.

Cornelissen, B., Zaidman, A., Deursen, A. van, Moonen, L., Koschke, R., (2009) A Systematic Survey of Program Comprehension through Dynamic Analysis. *IEEE Transactions on Software Engineering*. [online] 35 (5), 684–702.

Corubolo, F., Eggers, A.G., Hasan, A., Hedges, M., Waddington, S., Ludwig, J., (2014) '*A pragmatic approach to signifcant environment information collection to support object reuse*', in Proceedings of iPRES 2014 Melbourne, Australia.

Craig-Martin, M., 2003. *Becoming*. In collection of Tate, London (T11812)

Cramer, F. (2002) 'Concepts, notations, software, art', in *Seminar for Allegmeine und Vergleischende Literaturwissenschaft*. 2002 p. [online]. Available from: http://people.zhdk.ch/shusha.niederberger/doks/kunst-und-internet/cramer-concepts_notations_software_art_2002-clean.pdf (Accessed 11 May 2017).

Crnkovic, G. (2010) Constructive research and info-computational knowledge generation. *Model-Based Reasoning in Science and Technology*. 314359–380.

Crook, J. (2001) *Guide to Good Practice: Artists' Interviews*.

Crook, J. (2015) *Technique and Condition Texts: Writing Guide*.

Dappert, A., Guenther, R.S., Peyrard, S., (2016) *Digital Preservation Metadata for Practitioners: Implementing PREMIS*. Springer.

Dappert, A. & Farquhar, A. (2009) 'Significance is in the eye of the stakeholder', in *International Conference on Theory and Practice of Digital Libraries*. 2009 Springer. pp. 297–308. [online]. Available from: http://link.springer.com/10.1007/978-3-642-04346-8_29 (Accessed 10 August 2016).

Darling, P. W. (1985) Preservation vs. Conservation. *Abbey Newsletter 9 (6)*. [online]. Available from: http://cool.conservation-us.org/byorg/abbey/an/an09/an09-6/an09-604.html (Accessed 17 June 2017).

Das, S., Lutters, W.G., Seaman, C.B., (2007) 'Understanding documentation value in software maintenance', in *Proceedings of the 2007 Symposium on Computer human interaction for the management of information technology*. 2007 ACM. p. 2. [online]. Available from: http://dl.acm.org/citation.cfm?id=1234790.

Day, M. W. (2000) *Preservation of electronic information: a bibliography.* [online]. Available from: https://www.webarchive.org.uk/wayback/en/archive/20170705065345/http://homes.ukoln.ac.uk/~lismd/preservation.html.

David, C. (1997) *Politics-Poetics: Documenta X: The Book*. Cantz Verlag.

Davies, D. (2004) *Art as Performance*. Blackwell Publishing Ltd.

Day, R. E. (2016) All that is the Case: Documents and Indexicality. *Scribe*. 22 (1). [online].

Dekker, A. (2014) *Enabling the Future, or How to Survive FOREVER1: A study of networks, processes and ambiguity in net art and the need for an expanded practice of conservation*. PhD thesis.

Dekker, A. (2013) 'Enjoying the Gap: Comparing Contemporary Documentation on Strategies', in *Preserving and Exhibiting Media Art: Challenges and Perspectives*. Amsterdam University Press. pp. 150–169.

Depocas, A., Ippolito, J., Jones, C. (eds.) (2003) *Permanence Through Change: The Variable Media Approach*. New York, USA and Montreal, Canada: Guggenheim Museum Publications and The Daniel Langlois Foundation for Art, Science, and Technology. [online]. Available from: http://www.variablemedia.net/e/preserving/html/var_pub_index.html.

Derevenets, Y. (2017) *Snowman*. [online]. Available from: https://derevenets.com/ (Accessed 14 February 2018).

Dietrich, D. & Adelstein, F. (2015) Archival science, digital forensics, and new media art. *Digital Investigation*. [Online] 14, Supplement 1S137–S145.

Dietz, S. (2014) 'Collecting new-media art: Just like anything else, only different', in Bruce Altshuler (ed.) *New Collecting: Exhibiting and Audiences After New Media Art*. pp. 57–71.

Dietz, S. (2000) *Signal or Noise? The Network Museum* [online]. Available from: https://web.archive.org/web/20101020121400/https://walkerart.org/gallery9/webwalker/ww_032300.html (Accessed 4 July 2017).

Digital Preservation Coalition (2015) *Digital Preservation Handbook, 2nd Edition* [online]. Available from: http://handbook.dpconline.org/ (Accessed 21 June 2017).

Dipple, K., Laurenson, P., Fadenza-Rodrigues, F., (2010) '*Describing Networked Art for the Purpose of Documentation and Conservation*'.

DOCAM (n.d.) *DOCAM Documentation Model* [online]. Available from: http://www.docam.ca/en/documentation-model.html (Accessed 31 July 2017).

DOCAM (n.d.) *DOCAM Documentation Model: Typology of documents* [online]. Available from: http://www.docam.ca/en/typology-of-documents.html (Accessed 1 August 2017).

DOCAM (n.d.) *The DOCAM Research Alliance* [online]. Available from: http://www.docam.ca/ (Accessed 29 July 2017).

Dover, C. (2016) *How the Guggenheim and NYU Are Conserving Computer-Based Art. Guggenheim* [online]. Available from: https://www.guggenheim.org/blogs/checklist/how-the-guggenheim-and-nyu-are-conserving-computer-based-art-part-1 (Accessed 27 July 2017).

Dovi, S. (2017) 'Political Representation', in Edward N. Zalta (ed.) *The Stanford Encyclopedia of Philosophy*. Spring 2017 Metaphysics Research Lab, Stanford University. p. [online]. Available from: https://plato.stanford.edu/archives/spr2017/entries/political-representation/ (Accessed 15 February 2017).

Dreher, T. (2014) *History of Computer Art*. 1st Update (September 2015). [online]. Available from: http://iasl.uni-muenchen.de/links/GCA_Indexe.html.

Dresch, A., Lacerda, D.P., Antunes Jr, J.A.V., (2015) 'Design science research', in *Design Science Research*. Springer. pp. 67–102.

Duncan, W. (2009) *Making Ontological Sense of Hardware and Software*. [online]. Available from: http://www.cse.buffalo.edu/~rapaport/584/S10/duncan09-HWSWOnt.pdf (Accessed 27 July 2015).

Dupuy, E. (2017) *Java Decompiler*. [online]. Available from: http://jd.benow.ca/ (Accessed 8 September 2017).

Duranti, L. & Franks, P. C. (2015) *Encyclopedia of Archival Science*. Rowman & Littlefield.

Electronic Media Group (2015) TechFocus III: Caring for Software-based Art [online]. Available from: http://resources.conservation-us.org/techfocus/techfocus-iii-caring-for-computer-based-art-software-tw/ (Accessed 6 March 2019).

Eilam, E. (2011) *Reversing: secrets of reverse engineering.* John Wiley & Sons.

Enge, J. & Lurk, T. (2014) Classification and indexing of complex digital objects with CIDOC CRM. *Archiving Conference*. 2014 (1), 58–62.

Enge, J. & Lurk, T. (2013) 'Operational Practices for a Digital Preservation and Restoration Protocol', in *Preserving and Exhibiting Media Art. Challenges and Perspectives*. Amsterdam University Press. pp. 270–281.

Engel, D. & Hellar, M. (2014) *Technical Narratives and Software-Based Artworks*. [online]. Available from: http://www.si.edu/tbma/symposiums.

Engel, D. & Wharton, G. (2014) Reading between the lines: Source code documentation as a conservation strategy for software-based art. *Studies in Conservation*. [Online] 59 (6), 404–415.

Engel, D. & Wharton, G. (2015) Source Code Analysis as Technical Art History. *Journal of the American Institute for Conservation*. 54 (2), 91–101.

Ensom, T. (2018) *Software-based Artwork Structure Ontology*. [online]. Available from: https://github.com/tomensom/saso (Accessed 2 August 2018).

Falcão, P. (2010) *Developing a Risk Assessment Tool for the conservation of software-based artworks*. MA thesis. Bern. [online]. Available from: http://www.academia.edu/6660777/Developing_a_Risk_Assessment_Tool_for_the_conservation_of_software-_based_artworks_MA-Thesis.

Falcão, P. (2015) *John Gerrard Studio Visit Interview*. [audio recording].

Falcão, P. (2013) *Comparison of the Significant Properties of Software and Software-based Arts*. [unpublished document].

Falcão, P., Alistair, A., Jones, B., (2014) '*Virtualisation as a Tool for the Conservation of Software-Based Artworks*', Proceedings of iPRES 2014 Melbourne, Australia. [online]. Available from: https://www.academia.edu/12462584/Virtualisation_as_a_Tool_for_the_Conservation_of_Software-Based_Artworks (Accessed 19 May 2015).

Falcão, P. & Dekker, A. (2015) *Virtualizing John Gerrard's 'Sow Farm' (2009), or not?* [online]. Available from: https://vimeo.com/147884591.

Fauconnier, S. & Frommé, R. (2003) Capturing Unstable Media: Summary of research. [online]. Available from: http://v2.nl/files/2003/publishing/articles/capturing_summary.pdf.

Fernández, D.M., Böhm, W., Vogelsang, A., Mund, J., Broy, M., Kuhrmann, M., Weyer, T., (2018) Artefacts in Software Engineering: What are they after all? Preprint submitted to the *International Journal on Software and Systems Modeling* [Preprint]. Available from: http://arxiv.org/abs/1806.00098 (Accessed 30 July 2018).

Fino-Radin, B. (2016) *Art In the Age of Obsolescence* [online]. Available from: https://stories.moma.org/art-in-the-age-of-obsolescence-1272f1b9b92e (Accessed 9 February 2018).

Fino-Radin, B. (2018) Digital Art Storage: What Every Conservator Needs to Know. AIC News 43 (1). [online]. Available from: http://resources.conservation-us.org/aic-news/digital-art-storage-what-every-conservator-needs-to-know/ (Accessed 8 February 2018).

Fino-Radin, B. (2011) Digital Preservation Practices and the Rhizome Artbase. *Rhizome.org*.

Firesmith, D. (2007) Common Requirements Problems, Their Negative Consequences, and the Industry Best Practices to Help Solve Them. *Journal of Object Technology*. 6 (1), 17–33.

Gamarra, S., 2005. *LiMac Museum Shop*. In collection of Tate, London

Garijo, D. (2018) *Widoco*. [online]. Available from: https://github.com/dgarijo/Widoco.

Geffner, J. (2014) *What's the difference between a disassembler, debugger and decompiler?* [online]. Available from: http://reverseengineering.stackexchange.com/questions/4635/whats-the-difference-between-a-disassembler-debugger-and-decompiler (Accessed 4 October 2016).

Gerrard, J. (2015) *John Gerrard interviewed by Nicholas Forrest* [online]. Available from: http://uk.blouinartinfo.com/news/story/1103413/interview-john-gerrard-on-his-slippery-sims-at-thomas-dane (Accessed 31 January 2018).

Gerrard, J., 2009. *Sow Farm (near Libbey, Oklahoma) 2009.* In collection of Tate, London (T14279)

Gerrard, J. & Pötzelberger, W. (2015) *John Gerrard Studio Visit Interview.* Copy in Conservation Folder for artwork T14279. Tate, London.

Giaretta, D., Matthews, B., Bicarregui, J., Lambert, S., Guercio, M., Michetti, G., Sawyer, D., (2009) Significant Properties, Authenticity, Provenance, Representation Information

and OAIS Information. *California Digital Library*. [online]. Available from: http://escholarship.org/uc/item/0wf3j9cw (Accessed 26 May 2016).

Glinz, M. (2007) 'On non-functional requirements', in *Requirements Engineering Conference, 2007. RE'07. 15th IEEE International*. 2007 IEEE. pp. 21–26. [online]. Available from: http://ieeexplore.ieee.org/abstract/document/4384163/.

Goldstein, A. M. (2014) *Expert Eye: Bitforms Gallery's Steven Sacks on How to Collect New Media Art* [online]. Available from: http://www.artspace.com/magazine/interviews_features/how_to_collect_new_media_art (Accessed 25 November 2014).

Goodman, N. (1968) *Languages of art: An approach to a theory of symbols*. Hackett publishing.

Gordon, R. (2013) Material Significance in Contemporary Art. *ArtMatters: International Journal for Technical Art History*. 51–10.

Gordon, R. & Hermens, E. (2013) The Artist's Intent in Flux. *CeROArt. Conservation, exposition, Restauration d'Objets d'Art*. (HS), . [online]. Available from: http://journals.openedition.org/ceroart/3527 (Accessed 27 February 2018).

Gosain, A. & Sharma, G. (2015) 'A Survey of Dynamic Program Analysis Techniques and Tools', in *Proceedings of the 3rd International Conference on Frontiers of Intelligent Computing: Theory and Applications (FICTA) 2014*. Advances in Intelligent Systems and Computing. Springer. pp. 113–122. [online]. Available from: https://link.springer.com/chapter/10.1007/978-3-319-11933-5_13 (Accessed 18 February 2018).

Gould, S. J. & Eldredge, N. (1972) 'Punctuated Equilibria: An Alternative to Phyletic Gradualism', in Thomas J. M. Schopf (ed.) *Models in Paleobiology*. San Francisco, USA: Freeman, Cooper and Company. pp. 82–115.

Graham, B. & Cook, S. (2010) *Rethinking Curating: Art after New Media*. Cambridge, MA and London, England: The MIT Press.

Greenberg, J. (2005) Understanding Metadata and Metadata Schemes. *Cataloging & Classification Quarterly* 40 (3–4), 17–36. [online].

Greene, R. (2004) *Internet Art*. Thames & Hudson London.

Griesinger, P. (2016) Process history metadata for time-based media artworks at the Museum of Modern Art, New York. *Journal of Digital Media Management*. 4 (4), 331–342.

Guez, E., Stricot, M., Broye, L., Bizet, S., (2017) The afterlives of network-based artworks. *Journal of the Institute of Conservation*. [Online] 40 (2), 105–120.

Guttag, J. V. (2013) *Introduction to computation and programming using Python*. Mit Press.

Hagedoorn, H. (2017) *RivaTuner Statistics Server*. Guru3D. [online]. Available from: http://www.guru3d.com/files-details/rtss-rivatuner-statistics-server-download.html (Accessed 3 August 2017).

Haidvogl, M. (2015) *Acquiring and Documenting Jürg Lehni's 'Viktor' (2006~)*. [online]. Available from: https://vimeo.com/146980154. [online].

Haigh, T. (2011) The History of Information Technology. *Annual Review of Information Science and Technology*. 45 (1), 431–487. [online].

Hamilton, J. & Danicic, S. (2009) 'An evaluation of current java bytecode decompilers', in *Source Code Analysis and Manipulation, 2009. SCAM'09. Ninth IEEE International Working Conference on*. 2009 IEEE. pp. 129–136. [online]. Available from: http://ieeexplore.ieee.org/abstract/document/5279917/.

Hellar, M. (2013) *Smithsonian Institution Time-Based and Digital Art Working Group: Interview with Mark Hellar*. [online]. Available from: https://www.si.edu/content/tbma/documents/transcripts/MarkHellar_130614.pdf (Accessed 12 March 2018). [online]. Available from: https://www.si.edu/content/tbma/documents/transcripts/MarkHellar_130614.pdf (Accessed 12 March 2018).

Henry, L. J. (1998) Schellenberg in Cyberspace. *The American Archivist*. [Online] 61 (2), 309–327.

Hermens, E. (2012) 'Technical art history: the synergy of art, conservation and science', in Lenain, T., Locher, H., Pinotti, A., Rampley, M., Schoell-Glass, C., Zijlmans, K. (eds.) *Art History and Visual Studies in Europe: Transnational Discourses and National Frameworks*. Leiden, The Netherlands: Brill. p. [online]. Available from: http://eprints.gla.ac.uk/46122/ (Accessed 19 April 2015).

Hermens, E. & Fiske, T. (eds.) (2009) *Art, Conservation and Authenticities: Material, Concept, Context*. Archetype.

Herraiz, I., Rodriguez, D., Robles, G., Gonzalez-Barahona, J.M., (2013) The evolution of the laws of software evolution: A discussion based on a systematic literature review. *ACM Computing Surveys (CSUR)*. 46 (2), 28.

Heslop, H., Davis, S., Wilson, A., (2002) *An Approach to the Preservation of Digital Records*. [online]. Available from: https://www.ltu.se/cms_fs/1.83844!/file/An_approach_Preservation_dig_records.pdf.

Heydenreich, G. (2011) 'Documentation of Change–Change of Documentation', in *Inside Installations. Theory and Practice in the Care of Complex Artworks*. Amsterdam University Press. pp. 155–171.

Higgins, S. (2008) The DCC Curation Lifecycle Model. *International Journal of Digital Curation*. [Online] 3 (1), 134–140.

Hilton, P. (2016) *Where to start documenting a legacy system.* Writing by Peter Hilton. [online]. Available from: http://hilton.org.uk/blog/legacy-system-documentation (Accessed 25 July 2017).

Hiser, S. (2007) *Achieving Openness: a closer look at ODF & OOXML*. [online]. Available from: https://web.archive.org/web/20100612230613/http://odfalliance.org/resources/Achieving_Openness%20w-banner.pdf (Accessed 30 July 2018).

Hoggett, R. (2017) *1956 - CYSP-1 - Nicolas Schöffer - (Hungarian/French)* [online]. Available from: http://cyberneticzoo.com/cyberneticanimals/1956-cysp-1-nicolas-schoffer-hungarianfrench/ (Accessed 20 June 2017).

Hölling, H. (2017) The technique of conservation: on realms of theory and cultures of practice. *Journal of the Institute of Conservation*. [Online] 40 (2), 87–96.

IASA Technical Committee: Standards, Recommended Practices, and Strategies (2018) Guidelines for the Preservation of Video Recordings (IASA-TC 06). [online]. Available from: https://www.iasa-web.org/tc06/guidelines-preservation-video-recordings (Accessed 3 March 2019).

IEEE Computer Society (2014) *SWEBOK v3.0: Guide to the Software Engineering Body of Knowledge*. 3rd edition. Pierre Bourque & Richard E. Fairley (eds.). Los Alamitos, CA: IEEE Computer Society Press.

IFLA Study Group on the Functional Requirements for Bibliographic Records (2009) *Functional Requirements for Bibliographic Records: Final Report*.

INCCA (n.d.) Symposium: Modern Art: Who Cares? (1997) [online]. Available from: https://www.incca.org/events/symposium-modern-art-who-cares-1997 (Accessed 25 February 2019).

Inayat, I., Salim, S.S., Marczak, S., Daneva, M., Shamshirband, S., (2015) A systematic literature review on agile requirements engineering practices and challenges. *Computers in Human Behavior*. [Online]. 51 (Part B), 915–929.

Ippolito, J. (2003) 'Accommodating the Unpredictable: The Variable Media Questionnaire', in Depocas, A., Ippolito, J., Jones, C. (eds.) *Permanence Through Change: The Variable Media Approach*. New York, USA and Montreal, Canada: Guggenheim Museum Publications and The Daniel Langlois Foundation for Art, Science, and Technology. pp. 47–54. [online]. Available from: http://www.variablemedia.net/e/preserving/html/var_pub_index.html.

Ippolito, J. (2008) *Death by Wall Label* [online]. Available from: http://vectors.usc.edu/thoughtmesh/publish/11.php (Accessed 9 February 2018).

Jarczyk, A. (2015) *The Documentation of the Audiovisual Output and Interactive Experience*. [online]. Available from: https://vimeo.com/149089331.

Jazdzewski, C. (2014) *Why can't native machine code be easily decompiled?* [online]. Available from: https://softwareengineering.stackexchange.com/questions/229761/why-cant-native-machine-code-be-easily-decompiled (Accessed 30 July 2018).

Jodi (1997) *debate: dx webprojects* [online]. Available from: https://web.archive.org/web/20170611223729/http://www.documenta12.de/archiv/dx/lists/debate/0010.html (Accessed 11 June 2017).

John, J. L. (2012) *Digital Forensics and Preservation*. [online]. Available from: http://www.dpconline.org/component/docman/doc_download/810-dpctw12-03pdf (Accessed 3 October 2014).

Johnson, A. (2016*) How MediaWiki is streamlining San Francisco's new Museum of Modern Art*. Wikimedia Blog. [online]. Available from: https://blog.wikimedia.org/2016/07/07/sfmoma-mediawiki/ (Accessed 11 March 2018).

Jones, C. (2008) *Surveying the state of the art (of documentation).* [online]. Available from: http://www.fondation-langlois.org/html/e/page.php?NumPage=2126 (Accessed 17 July 2017).

JPEXS (2016) *JPEXS Free Flash Decompiler*. JPEXS. [online]. Available from: https://www.free-decompiler.com/flash/ (Accessed 4 October 2016).

Kaltman, E., Wardrip-Fruin, N., Lowood, H., Caldwell, C. (2014) *A Unified Approach to Preserving Cultural Software Objects and their Development Histories.* [online]. Available from: http://escholarship.org/uc/item/0wg4w6b9 (Accessed 16 March 2015).

Karch, E. (2011) *The Software Crisis: A Brief Look at How Rework Shaped the Evolution of Software Methodolgies* [online]. Available from: https://blogs.msdn.microsoft.com/karchworld_identity/2011/04/04/the-software-crisis-a-brief-look-at-how-rework-shaped-the-evolution-of-software-methodolgies/ (Accessed 18 August 2017).

Kasanen, E., Lukka, K., Siitonen, A., (1993) The constructive approach in management accounting research. *Journal of Management Accounting Research*. 5, 243.

Kay, A. & Goldberg, A. (1977) Personal Dynamic Media. *Computer*. 10 (3), 31–41.

Kirschenbaum, M., Ovenden, R., Redwine, G., Donahue, R., (2010) *Digital forensics and born-digital content in cultural heritage collections*. [online]. Available from: http://drum.lib.umd.edu/handle/1903/14722 (Accessed 17 December 2014).

Kirschenbaum, M. G. (2012) *Mechanisms: new media and the forensic imagination*. Cambridge, Mass.; London: MIT Press.

Knight, G. (2009) *InSPECT - Framework Report - Investigating Significant Properties of Electronic Content*. [online]. Available from: http://www.significantproperties.org.uk/inspect-framework.html (Accessed 11 November 2014).

Kopytoff, I. (1986) The cultural biography of things: commoditization as process. *The social life of things: Commodities in cultural perspective*. 6870–73.

Konstantelos, L., Delve, J., Anderson, D., Billenness, C., Baker, D., Dobreva, M. (Eds.), 2012. *The Preservation of Complex Objects Volume 2: Software Art*. [online]. Available from: http://www.pocos.org/books/pocos_vol_2.pdf (Accessed 3 October 2014).

Lagos, N., Waddington, S., Vion-Dury, J.-Y., (2015) 'On the Preservation of Evolving Digital Content – The Continuum Approach and Relevant Metadata Models', in *Metadata and Semantics Research*. Communications in Computer and Information Science. Springer, Cham. pp. 15–26. [online]. Available from: https://link.springer.com/chapter/10.1007/978-3-319-24129-6_2 (Accessed 19 July 2017).

Lambert, N. (2003) *A critical examination of computer art: its history and application*. University of Oxford. [online]. Available from: http://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.273456 (Accessed 14 October 2015).

Lambert, N. (2010) 'The Computer as a Dynamic Medium', in *Proceedings of the 1st international conference on Ideas before their time: connecting the past and present in computer art*. 2010 BCS Learning & Development Ltd. pp. 86–97.

Laposky, B. F. (1969) Oscillons: electronic abstractions. *Leonardo*. 345–354.

Laurenson, P. (2006) Authenticity, change and loss in the conservation of time-based media installations. *Tate Papers Autumn 2006*. [online]. Available from: http://www.tate.org.uk/research/publications/tate-papers/authenticity-change-and-loss-conservation-time-based-media.

Laurenson, P. (2010) *Time-based Media Conservation – Recent Developments from an Evolving Field*. [online]. Available from: https://vimeo.com/14632365 (Accessed 12 June 2017).

Laurenson, P. (2013) 'Old Media, New Media? Significant Difference and the Conservation of Software-Based Art', in *Preserving and Exhibiting Media Art. Challenges and Perspectives*. Amsterdam: Amsterdam University Press. pp. 73–96.

Laurenson, P. (2015) *The Lives of Digital Things: A Community of Practice Dialogue*. [online]. Available from: http://www.tate.org.uk/about-us/projects/pericles/lives-digital-things (Accessed 4 March 2018).

Laurenson, P. & van Saaze, V. (2014) Collecting Performance-based Art: New challenges and shifting perspectives. Performativity in the gallery: Staging interactive encounters. 27–41.

Lavington, S. H. (1998) Google-Books-ID: JRbESAAACAAJ. *A History of Manchester Computers*. Second Edition. British Computer Society.

Le Boeuf, P., Doerr, M., Emil Ore, C., Stead, S., (2015) *Definition of the CIDOC Conceptual Reference Model Version 6.1*.

Lehman, M. M. (1980) Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*. 68 (9), 1060–1076. [online].

Lehman, M. M. & Ramil, J. F. (2003) Software evolution—background, theory, practice. *Information Processing Letters*. 88 (1–2), 33–44.

Lehtiranta, L., Junnonen, J.-M., Kärnä, S., Pekuri, L., (2017) 'The Constructive Research Approach: Problem Solving for Complex Projects', in *Designs, Methods and Practices for Research of Project Management*. Online Edition Gower. pp. 95–106. [online]. Available from: https://web.archive.org/web/20161116225103/http://www.gpm-first.com:80/books/designs-methods-and-practices-research-project-management/constructive-research-approach.

Lethbridge, T.C., Singer, J., Forward, A., (2003) How software engineers use documentation: The state of the practice. *IEEE software*. 20 (6), 35–39.

Levy, D. M. (1994) *Fixed or Fluid? Document Stability and New Media.* [Online] 24–31.

Liu, L. & Özsu, M. T. (2009) *Encyclopedia of Database Systems*. Springer Publishing Company, Incorporated. [online]. Available from: http://dl.acm.org/citation.cfm?id=1804422 (Accessed 30 October 2015).

Lowood, H. (2013) 'The Lures of Software Preservation', in *Preserving.exe*. Library of Congress. p. [online]. Available from: http://www.digitalpreservation.gov/multimedia/documents/PreservingEXE_report_final101813.pdf.

Lozano-Hemmer, R. (2015) *Best practices for conservation of media art from an artist's perspective*. [online]. Available from: https://github.com/antimodular/Best-practices-for-conservation-of-media-art (Accessed 30 July 2017).

Lozano-Hemmer, R., (2005). *Subtitled Public.* In collection of Tate, London (T12565)

Lozano-Hemmer, R. (2006) *Subtitled Public Manual*. [online]. Available from: http://www.lozano-hemmer.com/texts/manuals/subPublic_manual.pdf (Accessed 21 July 2017).

Lurk, T. (2008) 'Virtualisation as conservation measure', in *Archiving Conference*. 2008 Society for Imaging Science and Technology. pp. 221–225.

Lurk, T., Espenschied, D., Enge, J. (2012) Emulation in the context of digital art and cultural heritage preservation. *PIK – Praxis der Informationsverarbeitung und Kommunikation.* 35 (4), 245–254.

Lynch, C. (2000) 'Authenticity and Integrity in the Digital Environment: An Exploratory Analysis of the Central Role of Trust', in *Authenticity in a Digital Environment*. Washington, D.C.: Council on Library and Information Resources. pp. 314–331.

Manchester, E. (2004) *Becoming - Summary* [online]. Available from: http://www.tate.org.uk/art/artworks/craig-martin-becoming-t11812 (Accessed 30 July 2017).

Manovich, L. (1996) *The Death of Computer Art.* Rhizome [online]. Available from: http://rhizome.org/community/41703/ (Accessed 18 May 2018).

Manovich, L. (2013) *Software Takes Command*. Bloomsbury Open Access Edition. Bloomsbury. [online]. Available from: http://dx.doi.org/10.5040/9781472544988.

Manovich, L. (2001) *The Language of New Media*. MIT Press.

Marchese, F. T. (2011) Conserving Digital Art for Deep Time. *Leonardo*. [Online] 44 (4), 302–308.

Marchese, F. T. (2013) 'Conserving software-based artwork through software engineering', in *Digital Heritage International Congress (DigitalHeritage), 2013*. [Online]. October 2013 pp. 181–184.

Marclay, C. (2010) *The Clock.* In collection of Tate, London (T14038)

Martinat Mendoza, J.C., 2007. *Brutalism: Stereo Reality Environment 3*. In collection of Tate, London (T13251)

Matters in Media Art (2015a) *About Matters in Media Art* [online]. Available from: http://mattersinmediaart.org/about.html (Accessed 29 July 2017).

Matters in Media Art (2015b) *Acquiring Media Art* [online]. Available from: http://mattersinmediaart.org/acquiring-time-based-media-art.html (Accessed 15 July 2017).

Matters in Media Art (2015c) *Documenting Media Art* [online]. Available from: http://mattersinmediaart.org/assessing-time-based-media-art.html (Accessed 17 July 2017).

Matters in Media Art (2015d) *Sustaining Media Art* [online]. Available from: http://mattersinmediaart.org/sustaining-your-collection.html (Accessed 24 January 2018).

Matthews, B., Shaon, A., Bicarregui, J., Jones, C., (2010) A framework for software preservation. *International Journal of Digital Curation*. 5 (1), 91–105.

Matthews, B., McIlwrath, B., Giaretta, D., Conway, E., (2008) *The Significant Properties of Software: A Study*.

Matthews, B., Shaon, A., Bicarregui, J., Jones, C., Woodcock, J., Conway, E., (2009) Towards a methodology for software preservation. *California Digital Library*.

Maxwell, J. A. (2005) 'Conceptual Framework: What Do You Think Is Going On?', in *Qualitative Research Design: An Interactive Approach*. SAGE. pp. 33–63.

McDonough, J.P., Kirschenbaum, M., Reside, D., Fraistat, N., Jerz, D., (2010) 'Twisty Little Passages Almost All Alike: Applying the FRBR Model to a Classic Computer Game.' *Digital Humanities Quarterly* 4 (2). [online]. Available from: http://www.digitalhumanities.org/dhq/vol/4/2/000089/000089.html#figure01 (Accessed 15 October 2015).

McDonough, J.P., Olendorf, R., Kirschenbaum, M., Kraus, K., Reside, D., Donahue, R., Phelps, A., Egert, C., Lowood, H., Rojo, S., (2010) *Preserving Virtual Worlds Final Report*. [online]. Available from: https://www.ideals.illinois.edu/handle/2142/17097 (Accessed 11 March 2015).

McGovern, N. Y. (2009) *Technology responsiveness for digital preservation: a model*. UCL (University College London). [online]. Available from: http://discovery.ucl.ac.uk/18017/1/18017.pdf.

McKemmish, S. (1994) 'Are records ever actual?', in *The Records Continuum: Ian Maclean and Australian Archives First Fifty Years*. Ancora Press. p. [online]. Available from: http://arrow.monash.edu.au/vital/access/services/Download/monash:155356/DOC.

McKemmish, S. (2001) Placing records continuum theory and practice. *Archival Science*. [Online] 1 (4), 333–359.

Mitzias, P., Kontopoulos, E., Riga, M., (2017) *Computer System Ontology Design Pattern* [online]. Available from: http://ontologydesignpatterns.org/wiki/Submissions:Computer_System (Accessed 16 February 2017).

Montfort, N. (2005) *Continuous Paper: The Early Materiality and Workings of Electronic Literature*. [online]. Available from: http://nickm.com/writing/essays/continuous_paper_mla.html (Accessed 8 November 2016).

Moor, J. H. (1978) Three Myths of Computer Science. *The British Journal for the Philosophy of Science*. 29 (3), 213–222.

Moser, A., Kruegel, C., Kirda, E., (2007) 'Limits of static analysis for malware detection', in *Computer security applications conference, 2007. ACSAC 2007. Twenty-third annual*. 2007 IEEE. pp. 421–430.

Muller, L. (2008) Towards an oral history of new media art. *Daniel Langlois Foundation*. [online]. Available from: http://www.fondation-langlois.org/pdf/e/towards-an-oral-history.pdf.

Muñoz-Viñas, S. (2004) *Contemporary Theory of Conservation*. 1 edition. Oxford; Burlington, MA: Routledge.

Munir, K. & Sheraz Anjum, M. (2018) The use of ontologies for effective knowledge modelling and information retrieval. *Applied Computing and Informatics.* [Online] 14 (2), 116–126.

Naeem, N.A., Batchelder, M., Hendren, L., (2007) 'Metrics for measuring the effectiveness of decompilers and obfuscators', in *Program Comprehension, 2007. ICPC'07. 15th IEEE International Conference on*. 2007 IEEE. pp. 253–258. [online]. Available from: http://ieeexplore.ieee.org/abstract/document/4268259/.

National Digital Information Infrastructure and Preservation Program (2013) Preserving.exe: Toward a National Strategy for Software Preservation. [online]. Available from: http://www.digitalpreservation.gov/multimedia/documents/PreservingEXE_report_final101813.pdf.

NDSA Infrastructure & Standards Working Groups (2014) *Checking Your Digital Content: How, What and When to Check Fixity?* [online]. Available from: https://blogs.loc.gov/thesignal/files/2014/02/NDSA-Checking-your-digital-content-Draft-2-5-14.pdf?loclr=blogsig (Accessed 8 February 2018).

Oberle, D., Grimm, S., Staab, S., (2009) 'An ontology for software', in *Handbook on ontologies*. Springer. pp. 383–402. [online]. Available from: http://link.springer.com/chapter/10.1007/978-3-540-92673-3_17/fulltext.html (Accessed 25 January 2017).

Object Management Group (2015) *Unified Modeling Language Version 2.5*. [online]. Available from: http://www.omg.org/spec/UML/2.5/.

Object Management Group (2005) *What is UML* [online]. Available from: http://www.uml.org/what-is-uml.htm (Accessed 9 September 2017).

Olson, M. (2012) *POSTINTERNET: Art After the Internet.* Foam magazine 29 p.59–63.

Paul, C. (2002) *CODeDOC* [online]. Available from: http://artport.whitney.org/commissions/codedoc/ (Accessed 29 January 2018).

Paul, C. (2003) *CODeDOC II* [online]. Available from: https://web.archive.org/web/20060821221233/http://www.aec.at:80/de/festival2003/programm/codedoc.asp (Accessed 29 January 2018).

Paul, C. (2015a) *Digital Art*. 3rd edition. London: Thames and Hudson Ltd.

Paul, C. (2015b) 'From Immateriality to Neomateriality: Art and the Conditions of Digital Materiality', in *Proceedings of the 21st International Symposium on Electronic Art*. 2015 p.

PERICLES Consortium & others (2014) *Deliverable 3.2: Linked Resource Model*. July.

Phillips, J. (2012) *Iteration Report*. [online]. Available from: https://www.guggenheim.org/wp-content/uploads/2015/11/guggenheim-conservation-iteration-report-2012.pdf.

Phillips, J. (2007) Reporting iterations: a documentation model for time-based media art. *Revista de História da Arte*. 4168–179.

Phillips, J., Engel, D., Dickson, E., Farbowitz, J., (2017) *Restoring Brandon, Shu Lea Cheang's Early Web Artwork.* Guggenheim [online]. Available from: https://www.guggenheim.org/blogs/checklist/restoring-brandon-shu-lea-cheangs-early-web-artwork (Accessed 10 February 2018).

Pistelli, D. (2012) *Explorer Suite*. NTCore. [online]. Available from: http://www.ntcore.com/exsuite.php (Accessed 14 February 2018).

Pitkin, H. F. (1967) *The Concept of Representation*. University of California Press.

Post, C. (2017) Preservation practices of new media artists: Challenges, strategies, and attitudes in the personal management of artworks. *Journal of Documentation*. [Online] 73 (4), 716–732.

PREMIS Editorial Committee & others (2015) PREMIS Data Dictionary for Preservation Metadata, version 3.0. *OCLC, Washington*.

Pressman, R. & Maxim, B. (2014) *Software Engineering: A Practitioner's Approach*. 8th Edition. New York, NY: McGraw-Hill Education.

Preston-Werner, T. (2013) *Semantic Versioning 2.0. 0*. [online]. Available from: https://semver.org/spec/v2.0.0.html.

Rayward, W. B. (1996) The History and Historiography of Information Science: Some Reflections. *Information Processing & Management*. 32 (1), 3–17.

Real, W. A. (2001) Toward Guidelines for Practice in the Preservation and Documentation of Technology-Based Installation Art. *Journal of the American Institute for Conservation.* [Online] 40 (3), 211–231.

Rechert, K., Espenschied, D., Valizada, I., Liebetraut, T., Russler, N., Suchodoletz, D. von, (2013) 'An Architecture for Community-Based Curation and Presentation of Complex Digital Objects', in Shalini R. Urs et al. (eds.) *Digital Libraries: Social Media and Community Networks*. Lecture Notes in Computer Science. Springer International Publishing. pp. 103–112. [online]. Available from: http://link.springer.com/chapter/10.1007/978-3-319-03599-4_12 (Accessed 17 November 2014).

Rechert, K., Falcão, P., Ensom, T., (2016) *Introduction to an emulation-based preservation strategy for software-based artworks*. [online]. Available from: http://www.tate.org.uk/research/publications/emulation-based-preservation-strategy-for-software-based-artworks (Accessed 23 March 2017).

Reed, B. (2005) Reading the records continuum: interpretations and explorations. *Archives and Manuscripts*. 33 (1), 18.

Rekoff, M. G. (1985) On reverse engineering. *IEEE Transactions on Systems, Man, and Cybernetics*. [online] SMC-15 (2), 244–252.

Rice, D. (2015) *Sustaining Consistent Video Presentation* [online]. Available from: http://www.tate.org.uk/about-us/projects/pericles/sustaining-consistent-video-presentation (Accessed 30 January 2018).

Rieger, O.Y., Murray, T., Casad, M., Alexander, D., Dietrich, D., Kovari, J., Muller, L., Paolillo, M., Mericle, D.K., (2015) *Preserving and Emulating Digital Art Objects*. [online]. Available from: http://ecommons.cornell.edu/handle/1813/41368 (Accessed 4 April 2016).

Rinehart, R. (2004) '*A System of Formal Notation for Scoring Works of Digital and Variable Media Art*', in Annual Meeting of the American Institute for Conservation of Historic and Artistic Works 2004, 14 June 2004 Portland, Oregon. [online]. Available from: http://cool.conservation-us.org/coolaic/sg/emg/library/pdf/rinehart/Rinehart-EMG2004.pdf (Accessed 30 March 2015).

Rinehart, R. (2007) The Media Art Notation System: Documenting and Preserving Digital/Media Art. *Leonardo*. 40 (2), 181–187. [online].

Rinehart, R. & Ippolito, J. (2014) *Re-collection: Art, New Media, and Social Memory*. Cambridge, Massachusetts: MIT Press.

Rokeby, D. (2010) *David Rokeby: The Giver of Names.* [online]. Available from: http://davidrokeby.com/gon.html (Accessed 8 March 2018).

Rothenberg, J. (1995) Ensuring the Longevity of Digital Documents. *Scientific American.* 272 (1), 42–47.

Rothenberg, J. (2002) Avoiding Technological Quicksand: Finding a Viable Technical Foundation for Digital Preservation: A Report to the Council on Library and Information Resources. Available from: https://clir.wordpress.clir.org/wp-content/uploads/sites/6/pub77.pdf (Accessed 15 March 2018).

Rosenthal, D. S. (2015) *Emulation & Virtualization as Preservation Strategies*. [online]. Available from: https://mellon.org/media/filer_public/0c/3e/0c3eee7d-4166-4ba6-a767-6b42e6a1c2a7/rosenthal-emulation-2015.pdf (Accessed 31 May 2016).

Rosenthal, D. S. (2012) *Formats through time.* DSHR's Blog [online]. Available from: http://blog.dshr.org/2012/10/formats-through-time.html (Accessed 29 July 2017).

Roux, S. (2016) The Document: A Multiple Concept. *Proceedings from the Document Academy*. 3 (1), 10.

Russinovich, M. (2017) *Process Monitor*. Microsoft. [online]. Available from: https://docs.microsoft.com/en-us/sysinternals/downloads/procmon.

Sakrowski, R. & Dullaart, C. (2018) *net.artdatabase* [online]. Available from: http://net.artdatabase.org/ (Accessed 9 February 2018).

Samyn, M. (2008) *Postmortem: Tale of Tales' The Graveyard* [online]. Available from: http://www.gamasutra.com/view/feature/132258/postmortem_tale_of_tales_the_.php?print=1 (Accessed 21 November 2016).

Saracevic, T. (2017) 'Information Science', in *Encyclopedia of Library and Information Sciences*. 3rd Edition. Boca Raton: Taylor & Francis.

Scholte, T. & Wharton, G. (2011) Inside installations: theory and practice in the care of complex artworks. Amsterdam University Press. [online]. Available from: http://www.bcin.ca/Interface/openbcin.cgi?submit=submit&Chinkey=431451 (Accessed 22 September 2016).

Schreibman, S., Siemens, R., Unsworth, J. (eds.) (2004) *A Companion to Digital Humanities*. Oxford: Blackwell. [online]. Available from: http://www.digitalhumanities.org/companion/ (Accessed 14 February 2017).

Scott, D. A. (2015) Conservation and authenticity: Interactions and enquiries. *Studies in Conservation*. [online] 60 (5), 291–305.

Shanken, E. A. (2009) *Art and electronic media*. Phaidon Press, London.

Shanken, E. A. (2002a) Art in the Information Age: Technology and Conceptual Art. *Leonardo*. 35 (4), 433–438.

Shanken, E. A. (2002b) Cybernetics and art: cultural convergence in the 1960s. *From Energy to Information*. 155–177.

Silberschatz, A., Galvin, P.B., Gagne, G., (2014) *Operating System Concepts Essentials*. Second Edition. John Wiley & Sons, Inc.

Singer, J. (1998) 'Practices of software maintenance', in *Proceedings of the International Conference on Software Maintenance, 1998*. 1998 IEEE. pp. 139–145. [online]. Available from: http://ieeexplore.ieee.org/abstract/document/738502/.

Sommerville, I. (2015) *Software Engineering*. 10th Edition (Global Edition). Boston, Mass.; Amsterdam; Cape Town: Pearson Education.

de Souza, S.C.B., Anquetil, N., Oliveira, K.M. de, (2006) Which documentation for software maintenance? *Journal of the Brazilian Computer Society*. 12 (3), 31–44.

Spear, A. D. (2006) *Ontology for the twenty first century: An introduction with recommendations*. [online]. Available from: http://ifomis.uni-saarland.de/bfo/documents/manual.pdf (Accessed 16 February 2017). [online]. Available from: http://ifomis.uni-saarland.de/bfo/documents/manual.pdf (Accessed 16 February 2017).

Spolsky, J. (2008) *Why are the Microsoft Office file formats so complicated? (And some workarounds).* Joel on Software [online]. Available from: https://www.joelonsoftware.com/2008/02/19/why-are-the-microsoft-office-file-formats-so-complicated-and-some-workarounds/ (Accessed 6 September 2017).

Stanford Center for Biomedical Informatics Research (2016) *protégé.* [online]. Available from: https://protege.stanford.edu/products.php (Accessed 29 July 2018).

Stringer, E. T. (2013) *Action research*. Sage Publications.

Stroulia, E. & Systä, T. (2002) Dynamic analysis for reverse engineering and program understanding. *ACM SIGAPP Applied Computing Review*. 10 (1), 8–17.

Suber, P. (1988) What is software? *The Journal of Speculative Philosophy*. 89–119.

von Suchodoletz, D., Rechert, K., Valizada, I., (2013) Towards Emulation-as-a-Service: Cloud Services for Versatile Digital Object Access. *International Journal of Digital Curation*. 8 (1), 131–142.

Taylor, G. D. (2014) *When the Machine Made Art: The Troubled History of Computer Art*. Bloomsbury Publishing USA.

Tate (2017) *Time-based media – Art Term* [online]. Available from: https://www.tate.org.uk/art/art-terms/t/time-based-media (Accessed 25 February 2019).

Terras, M. (2005) Reading the readers: Modelling complex humanities processes to build cognitive systems. *Literary and Linguistic Computing*. 20 (1), 41–59.

The Institute of Conservation (2014) *The Institute of Conservation's Code of Conduct*. [online]. Available from: https://icon.org.uk/system/files/documents/icon_code_of_conduct.pdf (Accessed 8 March 2018).

Thibodeau, K. (2002) *Overview of Technological Approaches to Digital Preservation and Challenges in Coming Years*. [online]. Available from: https://web.archive.org/web/20160520092136/http://www.clir.org:80/pubs/reports/pub107/thibodeau.html (Accessed 11 November 2014).

Tilley, S.R., Müller, H.A., Orgun, M.A., (1992) 'Documenting software systems with views', in *Proceedings of the 10th annual international conference on Systems documentation*. 1992 ACM. pp. 211–219. [online]. Available from: http://dl.acm.org/citation.cfm?id=147033.

Time-Based Media and Digital Art Working Group (2014) TECHNOLOGY EXPERIMENTS IN ART: Conserving Software-Based Artworks [online]. Available from: https://www.si.edu/tbma/symposiums (Accessed 6 March 2019).

Tribe, M. & Jana, R. (2006) *New Media Art*. Taschen London and Cologne.

Upward, F. (1996) Structuring the records continuum (Series of two parts) Part 1: Post custodial principles and properties. *Archives and Manuscripts*. 24 (2), 268.

Upward, F. (1997) Structuring the records continuum (Series of two parts) Part 2: Structuration theory and recordkeeping. *Archives and Manuscripts*. 25 (1), 10.

V2_Institute for the Unstable Media (2004) *Capturing Unstable Media: Glossary*.

V2_Institute for the Unstable Media (2003a) *Deliverable 1.2: Documentation and capturing methods for unstable media arts*. [online]. Available from: http://v2.nl/archive/articles/documentation-and-capturing-methods-for-unstable-media-arts.

V2_Institute for the Unstable Media (2003b) *Deliverable 1.3: Description models for unstable media art*. [online]. Available from: http://v2.nl/archive/articles/documentation-and-capturing-methods-for-unstable-media-arts.

van de Vall, R., Hölling, H., Scholte, T., Stigter, S., (2011) *Reflections on a biographical approach to contemporary art conservation*. [online]. Available from: http://dare.uva.nl/record/434262 (Accessed 8 October 2015).

van de Vall, R. (2015) The Devil and the Details: The Ontology of Contemporary Art in Conservation Theory and Practice. *The British Journal of Aesthetics*. 55 (3), 285–302.

van Saaze, V. (2013) *Installation Art and the Museum : Presentation and Conservation of Changing Artworks*. Amsterdam University Press.

Victoria and Albert Museum (2011) *A History of Computer Art* [online]. Available from: http://www.vam.ac.uk/content/articles/a/computer-art-history/ (Accessed 1 June 2017).

VMware (2018) *VMware Workstation Pro 12*. VMware. [online]. Available from: https://my.vmware.com/en/web/vmware/info/slug/desktop_end_user_computing/vmware_workstation_pro/14_0 (Accessed 3 March 2018).

Waddington, S., Hedges, M., Riga, M., Mitzias, P., Kontopoulos, E., Kompatsiaris, I., Vion-Dury, J.-Y., Lagos, N., Darányi, S., Corubolo, F., others, (2016) PERICLES–Digital Preservation through Management of Change in Evolving Ecosystems. *The Success of European Projects using New Information and Communication Technologies*. 51.

Waters, D. & Garrett, J. (1996) Preserving Digital Information. Report of the Task Force on Archiving of Digital Information.

Wardrip-Fruin, N. & Montfort, N. (2003) *The New Media Reader*. MIT press.

Wharton, G. (2016) Artist intention and the conservation of contemporary art. *Objects Specialty Group Postprints*. 22. [online]. Available from: http://resources.conservation-us.org/osg-postprints/wp-content/uploads/sites/8/2015/05/osg022-01.pdf.

Wharton, G. & Molotch, H. (2009) 'The Challenge of Installation Art', in *Conservation: Principles, Dilemmas and Uncomfortable Truths*. 1st Edition. Amsterdam; Boston: London: Elsevier/Butterworth-Heinemann; In Association with the Victoria & Albert Museum. 210–222.

White Cube (2010) *Christian Marclay: The Clock, Mason's Yard 2010* [online]. Available from: http://whitecube.com/exhibitions/christian_marclay_the_clock_masons_yard_2010/ (Accessed 1 February 2018).

Wiley, C., Novitskova, K., Dullaart, C., Archey, K., Coburn, T., Cairns, S., Cornell, L., (2013) Beginnings + Ends. *frieze magazine.* (159) [online]. Available from: https://frieze.com/article/beginnings-ends (Accessed 11 June 2017).

Wilson, A. (2007) *Significant Properties Report*. [online]. Available from: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.109.7923&rep=rep1&type=pdf (Accessed 12 August 2016).

Woods, K., Lee, C.A., Garfinkel, S., (2011) '*Extending digital repository architectures to support disk image preservation and access*', in [Online]. 2011 ACM Press. p. 57. [online]. Available from: http://portal.acm.org/citation.cfm?doid=1998076.1998088 (Accessed 11 November 2014).

World Wide Web Consortium (2012) *OWL 2 Web Ontology Language Document Overview* [online]. Available from: http://www.w3.org/TR/2012/REC-owl2-overview-20121211/.

Yeo, G. (2010) 'Nothing is the same as something else': significant properties and notions of identity and originality. *Archival Science*. [Online] 10 (2), 85–116.

Yuill, S. (2008) 'Concurrent Versions System', in Matthew Fuller (ed.) *Software Studies: A Lexicon*. Cambridge, Massachusetts; London, England: The MIT Press.

Zhang, P. & Benjamin, R. I. (2007) Understanding information related fields: A conceptual framework. *Journal of the American Society for Information Science and Technology*. 58 (13), 1934–1947.

APPENDIX I: ARTWORK CASE STUDY DESCRIPTIONS

9.1. Case Studies

Seven software-based artwork case studies were selected as the focus of this research. These artworks are all part of the Tate collection and have been displayed at least once since acquisition. As such, they are already accompanied by a considerable body of documentation generated within the institution, in addition to the tacit knowledge which resides in the conservators and other individuals who have been involved in their care.

In this section I briefly introduce each artwork, with the intention of providing essential background to enable the works to be referenced within the rest of the text without the need to repeat basic descriptive information. The summaries provided in the following sections include:

- Description of the work as a conceptual whole and critical information regarding the context of the work
- Description of the technologies involved in the production and realisation of the work, focusing particularly on the software components

- Photographic or screen capture documentation of at least one realisation of the work

9.1.1. Michael Craig-Martin - Becoming (2003)

Becoming consists of a 2D animation generated in real-time by a software program, and presented on a wall-mounted LCD screen. The screen's bevel provides framing and conceals the computer on which the software runs. The animation features an assemblage of objects rendered in the style of Craig-Martin's signature line drawings. Eighteen vividly coloured objects (a chair, a pair of pliers, a tape cassette, a fan, a pitchfork, a sandal, a light bulb, a drawer, a metronome, a book, a bucket, a TV, a flashlight, a safety-pin, a knife, a pair of handcuffs and a medicine jar spilling pills) fade in and out of visibility against a fuchsia pink background. The number of objects visible at any one time is randomised so that unpredictable combinations may arise. This work is one of a series of technologically similar works created in collaboration with the London-based digital design and production studio AVCO.

Figure 27. Michael Craig-Martin, *Becoming*, 2003 (T11812). Photograph of installed work. © Michael Craig-Martin and Tate, London 2018.

Underlying *Becoming* is a Macromedia Shockwave executable file running on a Windows XP PC. The objects are 2D vector images positioned in a pre-defined relational arrangement, the rendering and animation of which is controlled by the Shockwave playback engine embedded in the executable. The images fade in an out of the screen according to parameters defined in the code, which constrains the number of objects visible at any one time and the speed and regularity with which they appear and disappear. The software was originally developed in Macromedia Director, with custom code written in the Lingo scripting language. As part of a research project undertaken in 2010, the software was ported to Flash, with the code rewritten for ActionScript 3.0.

9.1.2. Cory Arcangel - Colors (2005)

Colors is a software program which plays back the 1988 movie *Colors* (directed by Dennis Hopper), one line of horizontal pixels at a time, with each line stretched vertically to fill the screen. The resulting animated bands of colour are presented as a projection in a dark exhibition space, with the original movie soundtrack playing from stereo speakers. The dynamic patterns of abstract colour reference the source film itself (which is about Los Angeles gangs), the analog special effects technique known as slit-scan, and artistic practices such as colour field painting and experimental film.

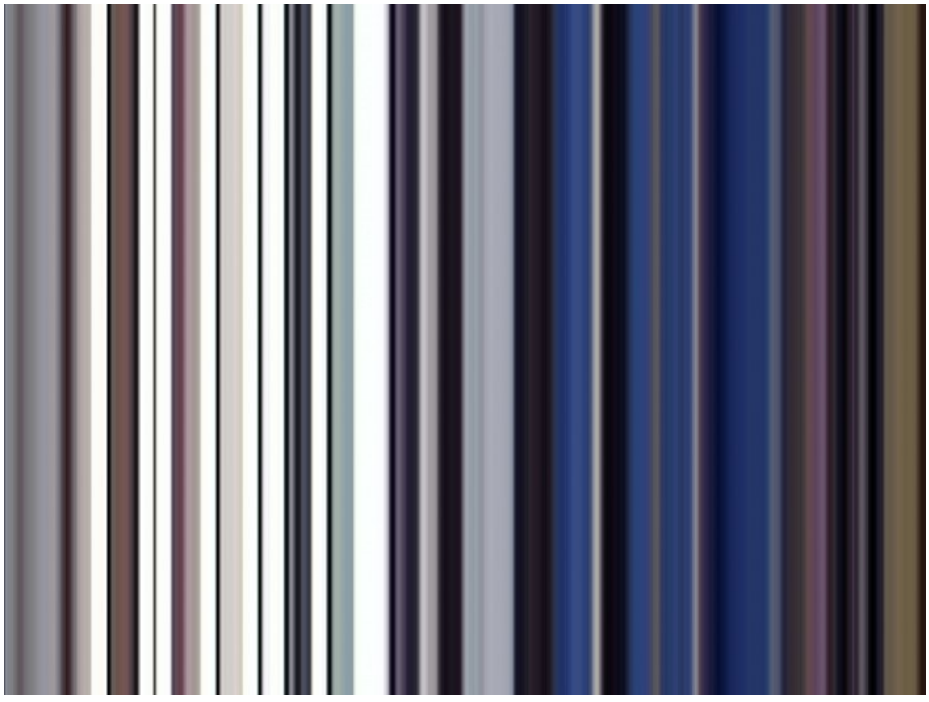


Figure 28. Cory Arcangel, *Colors*, 2005 (L02995). Still image capture. © Cory Arcangel and Tate, London 2018.

The software employed is an Apple OSX application developed in Apple's XCode development environment using the OpenFrameworks toolkit. The application uses the QuickTime and OpenGL frameworks embedded in OSX to process a QuickTime digital video file in real-time. The video file is sourced from a DVD of the original movie. The video file is played back frame by frame according to its encoded framerate, but only one horizontal row of pixels is rendered each time, and each row stretched vertically to fill a 1024x768 resolution area. The software loops after it has played the entire movie through for that pixel row, and moves on to the next row of horizontal pixels from the first frame. When the program reaches the last line of pixels, it returns to the first row and starts the process again.

9.1.3. Sandra Gamarra - LiMac Museum Shop (2005)

LiMac Museum Shop is an installation which mimics the formal trappings of a museum gift shop, and forms a part of a larger project in which Gamarra has created a fictional museum of contemporary art for Lima, Peru ('Museo de Arte Contemporáneo de Lima' or 'LiMac'). Gamarra has constructed a complete corporate identity for the museum including branding, merchandise and the focus for this case study: a website. The site is accessible via a terminal as part of the installation, but also exists externally and independently at a public domain, where it remains under the artists control and is regularly updated. The website includes online exhibitions, a shop and even a spurious "friends of the museum" scheme.

Figure 29. Sandra Gamarra, LiMac Museum Shop, 2005. Images of installation at Tate Modern in 2011. © Sandra Gamarra and Tate, London 2018.

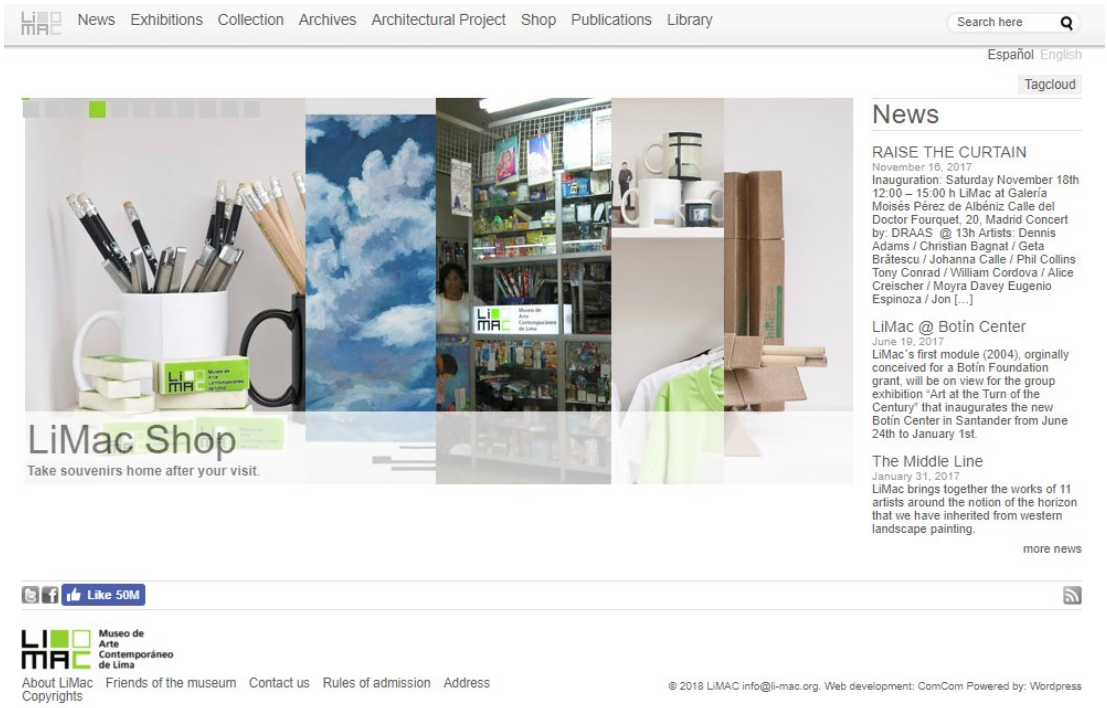


Figure 30. Screenshot of the Wordpress-based LiMac website in 2018. © Sandra Gamarra.

The LiMac website has been realised in various version through time, aping the progression of museum website development and design. The current version of the website was developed for WordPress by ComCom, a Spanish web design company. WordPress is a content management system (CMS) which provides a user-friendly website management and customisation interface, and has a back-end which supports custom PHP code for page templates. The WordPress installation is supported by a server running the LAMP stack, a popular server platform which consists of a Linux operating system, Apache Web Server software, MySQL database software and the PHP interpreter. These tools operate together to serve the web pages to site visitors via HTTP in their chosen web browser.

9.1.4. Rafael Lozano-Hemmer - Subtitled Public (2005)

Subtitled Public is an interactive installation which tracks visitors to a darkened exhibition space, and projects a 'subtitle' word onto them which follow them around the space. Visitors are monitored by surveillance cameras, which feed images to a motion-tracking software installed on a network of computers. The words are selected randomly from a pre-defined set of verbs conjugated in the third person. If two visitors touch, the words projected onto them will be exchanged. Occasionally the subtitling process is interrupted for a short time, as the camera feed is projected back into room.

Figure 31. Rafael Lozano-Hemmer, *Subtitled Public*, 2005 (T12565). Photograph of two subtitled gallery visitors interacting during an installation. © Rafael Lozano-Hemmer and Tate, London 2018.

Subtitled Public was developed in the Borland Delphi programming environment, which uses a derivative of the language Object Pascal. The programmer, Conroy Badger, utilised open source computer vision libraries, from which bespoke software was constructed to implement the tracking system. The software consists of a set of Windows executables which were developed for the Windows XP operating system, and installed on a set of Mac Mini computers running Bootcamp. The full expanse of the exhibition space is covered using a variable number of surveillance pods running this software, each consisting of a computer, infra-red sensitive camera and short throw projector. The surveillance pod computers are networked and controlled centrally by a master computer.

9.1.5. Jose Carlos Martinat Mendoza - Stereo Reality Environment 3: Brutalismo (2007)

At the centre of the *Stereo Reality Environment 3: Brutalismo* installation is a scale model of the former Peruvian military headquarters building known as the “Pentagonito”, which became notorious during the Fujimori presidency as a site of torture and murder perpetrated by the secret service. On top of the sculpture sit a set

of thermal printers, which are connected to a computer visible on the floor of the exhibition space. This computer is connected to the internet, and harvests fragments of text from web page which contain references to “Brutalismo” or “Brutalism”. These text fragments are then printed onto slips of paper which fall to the gallery floor and accumulate during the exhibition period.

Figure 32. Jose Carlos Martinat Mendoza, *Stereo Reality Environment 3: Brutalism*, 2007 (T13251). Photograph of the work installed at Tate Modern in 2011. © Jose Carlos Martinat Mendoza and Tate, London 2018.

Brutalism employs a pair Java programs, developed in the NetBeans IDE, which carry out two primary functions. The first carries out internet searches using the Google Search API, scrape fragments of text from search results for the terms ‘brutalism’ and ‘brutalismo’, and store these fragments in a MySQL database for later access. The second program takes text fragments from the database and prints them out via the thermal printers. The software runs from on a repurposed Dell workstation PC,

running the Linux Ubuntu operating system, which is visibly connected via a mass of cabling. The software originally connected to the till receipt printers employed via the DB-25 parallel port interface, while a later version was developed that can utilise the USB protocol.

9.1.6. John Gerrard - Sow Farm (near Libbey, Oklahoma) 2009 (2009)

Sow Farm (near Libbey, Oklahoma) 2009 is a real-time 3D simulation depicting an unmanned pig farm in a remote region of the Great Plains in Oklahoma, United States, seen from the perspective of a slowly circling virtual camera. Running a complete simulation cycle over a period of 365 days real time, the 3D environment features realistic rendering of industrial buildings, arid prairie landscape and day-night cycles complete with dynamic sun and stars. Once every 156 days in real time, a truck drives up to the buildings and waits for one hour. *Sow Farm* is one in a series of works depicting buildings relating to the military-industrial complex in the USA. The work can be displayed in a variety of ways, but is usually presented as a projection in a darkened gallery space.

Figure 33. John Gerrard, *Sow Farm (near Libbey, Oklahoma) 2009,* 2009 (T14279). Photograph of the work installed at Tate Britain in 2016. © John Gerrard and Tate, London 2018.

The *Sow Farm* software consists of a Windows executable file which packages the 3D data assets and rendering engine, and an associated set of plain-text

configuration files. Running the software at the level of quality specified by the artist requires a high performance PC with a powerful graphics card and access to Microsoft's DirectX 9 framework. The software was built in a proprietary software package for the development of real-time 3D applications called Quest3D. This software package allows the authoring of complex 3D environments (such as games or architectural simulations) without having to write a 3D rendering engine from scratch. Custom code components were added by the artist and his team in the form of Quest3D plugins (written in C++) and HLSL shaders.

APPENDIX II: CONCEPTUAL MODEL FOR THE REPRESENTATION OF SOFTWARE-BASED ARTWORK SYSTEMS

10.1. Introduction to OWL 2 Ontology

Below is documentation of the classes and object properties and data properties specified in version 1.00 of the Software-based Art Structure Ontology. Documentation was generated using Widoco (Garijo, 2018), and a RDF/XML format OWL 2 (World Wide Web Consortium, 2012) ontology generated in Protégé 5.2 (Stanford Center for Biomedical Informatics Research, 2016). Named individuals have been excluded from this version for brevity, but are available in the online version, maintained on GitHub by the author (Ensom, 2018). While the version of the ontology which this documentation describes will remain static as a part of this thesis, the GitHub version may be updated in the future, so should be referred to if the ontology is being reused.

The following system of annotation is used to indicate entity types:

- [c]: Classes
- [op]: Object Properties

- [dp]: Data Properties

10.2. Classes

Abstract Component[c]

IRI: http://tomensom.com/saso#AbstractComponent

has sub-classes
    Technical Environment [c]
is in domain of
    is externally hosted [dp]

Android[c]

IRI: http://tomensom.com/saso#Android

has super-classes
    Operating System [c]
is disjoint with
    Linux [c], MacOS [c], Windows [c], iOS [c]

API[c]

IRI: http://tomensom.com/saso#API

has super-classes
    Software [c]
has members
    google search a p i 2011 [ni]
is disjoint with
    Binary [c], Database Software [c], Driver [c], Instrument [c], Operating System [c], Runtime Environment [c], Runtime Library [c]

Artwork[c]

IRI: http://tomensom.com/saso#Artwork

A distinct intellectual creation.

is in domain of
    has realisation [op], has variant [op], has version [op]
has members
    t11812 becoming [ni], t13251 brutalismo [ni], t14279 sow farm [ni]

Audio Interface[c]

IRI: http://tomensom.com/saso#AudioInterface

has super-classes
    External Hardware [c]

Binary[c]

IRI: http://tomensom.com/saso#Binary

The executable representation of a software program.

has super-classes
    Software [c]
is disjoint with
    API [c], Database Software [c], Driver [c], Instrument [c], Operating System [c], Runtime Environment [c], Runtime Library [c]

Case[c]

IRI: http://tomensom.com/saso#Case

has super-classes
    Hardware [c]
is disjoint with
    External Hardware [c], Internal Hardware [c]

Concrete Component[c]

IRI: http://tomensom.com/saso#ConcreteComponent

has sub-classes
    Data [c], Hardware [c], Software [c]
is in domain of
    is externally hosted [dp]

Connector[c]

IRI: http://tomensom.com/saso#Connector

Software that allows communication between other software components.

has super-classes
    Software [c]
has members
    j d b c [ni]

Controller[c]

IRI: http://tomensom.com/saso#Controller

Hardware that provides an interface for other hardware to connect to the host machine.

has super-classes
    Internal Hardware [c]

CPU[c]

IRI: http://tomensom.com/saso#CPU

has super-classes
    Internal Hardware [c]
is disjoint with
    GPU [c], Internal Soundcard [c], RAM [c], Storage Device [c]

Data[c]

IRI: http://tomensom.com/saso#Data

A component with a material manifestation which is only verifiable through the use of computer hardware and appropriate rendering software.

has super-classes
    Concrete Component [c]
has sub-classes
    Image [c], SQL [c], Video [c]
is in range of
    has data component [op]

Database Software[c]

IRI: http://tomensom.com/saso#DatabaseSoftware

Software that manages databases.

has super-classes
    Software [c]
has members
    my s q l 5.1 [ni]
is disjoint with
    API [c], Binary [c], Driver [c], Instrument [c], Operating System [c], Runtime Environment [c], Runtime Library [c]

Display Device[c]

IRI: http://tomensom.com/saso#DisplayDevice

has super-classes
    External Hardware [c]
has sub-classes
    Monitor [c], Projector [c]
is disjoint with
    Keyboard [c], Mouse [c], Printer [c]

Driver[c]

IRI: http://tomensom.com/saso#Driver

A specialised form of software which supports communication between software, operating system and hardware.

has super-classes
Software [c]
has members
n v i d i a display driver 285.62 [ni]
is disjoint with
API [c], Binary [c], Database Software [c], Instrument [c], Operating System [c], Runtime Environment [c], Runtime Library [c]

External Hardware[c]

IRI: http://tomensom.com/saso#ExternalHardware

Hardware component which is intended to be exposed during use.

has super-classes
Hardware [c]
has sub-classes
Audio Interface [c], Display Device [c], Keyboard [c], Mouse [c], Printer [c]
is disjoint with
Case [c], Internal Hardware [c]

GPU[c]

IRI: http://tomensom.com/saso#GPU

has super-classes
Internal Hardware [c]
is disjoint with
CPU [c], Internal Soundcard [c], RAM [c], Storage Device [c]

Hardware[c]

IRI: http://tomensom.com/saso#Hardware

A component with a material manifestation which is verifiable without the use of other hardware.

has super-classes
Concrete Component [c]
has sub-classes
Case [c], External Hardware [c], Internal Hardware [c]
is in domain of
has interface [op], is virtual [dp]
is in range of
has hardware component [op], has interface [op]

Hardware Environment[c]

IRI: http://tomensom.com/saso#HardwareEnvironment

A constellation of interconnected software components that form an environment in which software might be executed.

has super-classes
Technical Environment [c]
is in domain of
has hardware component [op]
has members
t13251 brutalismo dell workstation [ni], t14279 sow farm custom p c1 [ni], t14279 sow farm v mware v m test [ni]

HDD[c]

IRI: http://tomensom.com/saso#HDD

has super-classes
Storage Device [c]
is disjoint with
SSD [c], SSHD [c]

Image[c]

IRI: http://tomensom.com/saso#Image

has super-classes
Data [c]

Instrument[c]

IRI: http://tomensom.com/saso#Instrument

A specialised type of software which is capable of intercepting or measuring the properties of a hardware or software component.

has super-classes
Software [c]
is disjoint with
API [c], Binary [c], Database Software [c], Driver [c], Operating System [c], Runtime Environment [c], Runtime Library [c]

Internal Hardware[c]

IRI: http://tomensom.com/saso#InternalHardware

Hardware component which is intended to be enclosed during use.

has super-classes
Hardware [c]
has sub-classes
CPU [c], Controller [c], GPU [c], Internal Soundcard [c], RAM [c], Storage Device [c]
is disjoint with
Case [c], External Hardware [c]

Internal Soundcard[c]

IRI: http://tomensom.com/saso#InternalSoundcard

has super-classes
Internal Hardware [c]
is disjoint with
CPU [c], GPU [c], RAM [c], Storage Device [c]

iOS[c]

IRI: http://tomensom.com/saso#iOS

has super-classes
Operating System [c]
is disjoint with
Android [c], Linux [c], MacOS [c], Windows [c]

Keyboard[c]

IRI: http://tomensom.com/saso#Keyboard

has super-classes
External Hardware [c]
is disjoint with
Display Device [c], Mouse [c], Printer [c]

Linux[c]

IRI: http://tomensom.com/saso#Linux

has super-classes
Operating System [c]
is disjoint with
Android [c], MacOS [c], Windows [c], iOS [c]

MacOS[c]

IRI: http://tomensom.com/saso#MacOS

has super-classes
Operating System [c]
is disjoint with
Android [c], Linux [c], Windows [c], iOS [c]

Monitor[c]

IRI: http://tomensom.com/saso#Monitor

has super-classes
Display Device [c]

Mouse[c]

IRI: http://tomensom.com/saso#Mouse

has super-classes
External Hardware [c]
is disjoint with
Display Device [c], Keyboard [c], Printer [c]

Operating System[c]

IRI: http://tomensom.com/saso#OperatingSystem

A specialised form of software supporting the execution of software programs and communication with hardware and other components. An operating system is usually composed of a kernel—the primary control system—and supporting interfaces, frameworks and services.

has super-classes
Software [c]
has sub-classes
Android [c], Linux [c], MacOS [c], Windows [c], iOS [c]
has members
ubuntu 7.04 [ni], windows 7 build7601 [ni]
is disjoint with
API [c], Binary [c], Database Software [c], Driver [c], Instrument [c], Runtime Environment [c], Runtime Library [c]

Printer[c]

IRI: http://tomensom.com/saso#Printer

has super-classes
External Hardware [c]
has members
thermal printer1 [ni], thermal printer2 [ni], thermal printer3 [ni], thermal printer4 [ni]
is disjoint with
Display Device [c], Keyboard [c], Mouse [c]

Projector[c]

IRI: http://tomensom.com/saso#Projector

has super-classes
Display Device [c]

RAM[c]

IRI: http://tomensom.com/saso#RAM

has super-classes
Internal Hardware [c]
is disjoint with
CPU [c], GPU [c], Internal Soundcard [c], Storage Device [c]

Realisation[c]

IRI: http://tomensom.com/saso#Realisation

An embodiment of a particular variant of the work in time and space.

is in domain of
    has constituent [op]
is in range of
    has realisation [op]
has members
    t11812 becoming realisation tate britain2013 [ni], t13251 brutalismo realisation2011 [ni], t14279 sow farm realisation2016 [ni]

Runtime Environment[c]

IRI: http://tomensom.com/saso#RuntimeEnvironment

Software that provides an environment in which other software can be executed.

has super-classes
    Software [c]
has members
    j r e 7 [ni]
is disjoint with
    API [c], Binary [c], Database Software [c], Driver [c], Instrument [c], Operating System [c], Runtime Library [c]

Runtime Library[c]

IRI: http://tomensom.com/saso#RuntimeLibrary

Software which provides shared functionality, usable by other software at runtime.

has super-classes
    Software [c]
has members
    direct x runtime april2005 x86 [ni], phidget21 library x86 [ni]
is disjoint with
    API [c], Binary [c], Database Software [c], Driver [c], Instrument [c], Operating System [c], Runtime Environment [c]

Software[c]

IRI: http://tomensom.com/saso#Software

A component with a material manifestation which is only verifiable through the use of computer hardware.

has super-classes
    Concrete Component [c]
has sub-classes
    API [c], Binary [c], Connector [c], Database Software [c], Driver [c], Instrument [c], Operating System [c], Runtime Environment [c], Runtime Library [c], Software Super-Object [c]

is in domain of
architecture [dp], has interface [op]
is in range of
has hardware component [op], has interface [op], has software component [op]

Software Environment[c]

IRI: http://tomensom.com/saso#SoftwareEnvironment

A constellation of interconnected hardware components that form an environment in which software might be executed.

has super-classes
Technical Environment [c]
is in domain of
has software component [op]
is in range of
hosts environment [op]
has members
t13251 brutalismo software environment1 [ni], t14279 sow farm software environment1 [ni]

Software Super-Object[c]

IRI: http://tomensom.com/saso#SoftwareSuperObject

A subset of software consisting of binaries and data assets which perform some function or purpose. This component is a simplification of what may be a very variable structure.

has super-classes
Software [c]
is in domain of
has data component [op], has software component [op], is executable in [op]
has members
t13251 brutalismo s s o [ni], t14279 sow farm s s o [ni]

SQL[c]

IRI: http://tomensom.com/saso#SQL

has super-classes
Data [c]
has members
t13251 brutalismo database [ni]

SSD[c]

IRI: http://tomensom.com/saso#SSD

has super-classes
Storage Device [c]
is disjoint with

HDD [c], SSHD [c]

SSHD[c]

IRI: http://tomensom.com/saso#SSHD

has super-classes
Storage Device [c]
is disjoint with
HDD [c], SSD [c]

Storage Device[c]

IRI: http://tomensom.com/saso#StorageDevice

is equivalent to
HDD [c] or SSD [c] or SSHD [c]
has super-classes
Internal Hardware [c]
has sub-classes
HDD [c], SSD [c], SSHD [c]
is disjoint with
CPU [c], GPU [c], Internal Soundcard [c], RAM [c]

Technical Environment[c]

IRI: http://tomensom.com/saso#TechnicalEnvironment

A constellation of interconnected hardware and software components that form an environment in which a software program might be executed.

is equivalent to
Hardware Environment [c] or Software Environment [c]
has super-classes
Abstract Component [c]
has sub-classes
Hardware Environment [c], Software Environment [c]
is in range of
is executable in [op]

Variant[c]

IRI: http://tomensom.com/saso#Variant

A specific implementation of a version which has broadly similar formal, functional and behavioural characteristics.

is in domain of
has realisation [op]
is in range of
has variant [op]
has members

t11812 becoming variant flash2010 [ni], t11812 becoming variant shockwave2003 [ni]

Version[c]

IRI: http://tomensom.com/saso#Version

An expression of the artwork with well defined formal, functional and behavioural characteristics.

is in domain of
has realisation [op], has variant [op]
is in range of
has version [op]
has members
t11812 becoming version 2003 [ni]

Video[c]

IRI: http://tomensom.com/saso#Video

has super-classes
Data [c]

Windows[c]

IRI: http://tomensom.com/saso#Windows

has super-classes
Operating System [c]
is disjoint with
Android [c], Linux [c], MacOS [c], iOS [c]

10.3. Object Properties

has constituent[op]

IRI: http://tomensom.com/saso#hasConstituent

A Realisation is made up of one or more Software Super-Object and one or more Technical Environment.

has domain
Realisation [c]
has range
Software Super-Object [c] or Technical Environment [c]

has data component[op]

IRI: http://tomensom.com/saso#hasDataComponent

A Software Super-Object can consist of one or more Data components.

has domain
    Software Super-Object [c]
has range
    Data [c]

has hardware component[op]

IRI: http://tomensom.com/saso#hasHardwareComponent

A Hardware Environment can consist of one or more Hardware components.

has domain
    Hardware Environment [c]
has range
    Hardware [c]
    Software [c]

has interface[op]

IRI: http://tomensom.com/saso#hasInterface

A Software or Hardware component may use a Software or Hardware component to communicate with another Software and Hardware component.

has domain
    Hardware [c]
    Software [c]
has range
    Hardware [c]
    Software [c]

has realisation[op]

IRI: http://tomensom.com/saso#hasRealisation

An Artwork, Version or Variant may have one or more Realisation.

has domain
    Artwork [c]
    Variant [c]
    Version [c]
has range
    Realisation [c]

has software component[op]

IRI: http://tomensom.com/saso#hasSoftwareComponent

A Software Super-Object or Software Environment can consist of one or more Software component.

has domain
    Software Environment [c]
    Software Super-Object [c]
has range
    Software [c]

has variant[op]

IRI: http://tomensom.com/saso#hasVariant

An Artwork or Version may have one or more Variant.

has domain
    Artwork [c]
    Version [c]
has range
    Variant [c]

has version[op]

IRI: http://tomensom.com/saso#hasVersion

An Artwork may have one or more Version.

has domain
    Artwork [c]
has range
    Version [c]

hosts environment[op]

IRI: http://tomensom.com/saso#hostsEnvironment

A Technical Environment (Hardware or Software) may host another Software Environment.

has domain
    Hardware Environment [c] or Software Environment [c]
has range
    Software Environment [c]

is executable in[op]

IRI: http://tomensom.com/saso#isExecutableIn

A Software Super-Object may have one or more Technical Environment in which it can be sucessfully executed.

has domain
    Software Super-Object [c]
has range
    Technical Environment [c]

10.4. Data Properties

architecture[dp]

IRI: http://tomensom.com/saso#architecture

Describes the processor or instruction set architecture that the software component is designed for.

has domain
    Software [c]

is externally hosted[dp]

IRI: http://tomensom.com/saso#isExternallyHosted

Indicates that a component is not maintained by the organisation.

has domain
    Abstract Component [c]
    Concrete Component [c]

is virtual[dp]

IRI: http://tomensom.com/saso#isVirtual

Indicates that a hardware component is virtual.

has domain
    Hardware [c]

version[dp]

IRI: http://tomensom.com/saso#version

The version number or code of a particular component.

has domain
    Data [c] or Software [c]

APPENDIX III: SOFTWARE-BASED ARTWORK TECHNIQUE AND CONDITION TEXTS

11.1. Introduction to Technique and Condition Texts

These texts were produced as part of my PhD research, using information gathered during the analysis of the artwork case studies. Each text is written to comply with Tate's guidelines on the writing of technical entries for artworks in the collection, while also necessitating a reconsideration of how these texts might be written in order to accommodate the specific conceptual and technical considerations posed by software-based art, and time-based media more generally.

It should be noted that the texts, as they appear here, are unedited drafts and are not necessarily representative of those that will be published as part of Tate's online collections information in the future. The LiMac case study is also excluded from the texts written, as the website focused on in this thesis is only one component of the installation in the Tate collection, and a lack of direct access to the technical components of the work (which are managed by the artist) prevents the analysis required to write such a text.

11.2 Michael Craig-Martin - Becoming (2003)

Becoming consists of custom software used to generate dynamic 2D graphics displayed on an LCD screen. These graphics are comprised of a set of brightly coloured images of household objects which fade in and out in randomly against a fuchsia background. The screen is embedded within a wall-mounted grey and black case, which conceals the computer hardware on which the software runs. The software itself is a Windows Portable Executable file, which contains custom code, 2D graphics and Shockwave playback functionality. The executable file–also known as a Shockwave projector–does not require any external data or supporting software beyond the Windows XP operating system on which it runs.

The software was developed in 2003 using Macromedia Director 8, a tool for creating Shockwave multimedia applications. At this time, Macromedia Director 8 was a commercial tool in widespread use for creating multimedia content for digital platforms. This work is one of the first works by Craig-Martin to use this technology and was produced by Daniel Jackson at the London-based digital design company AVCO Productions. AVCO also worked with other prominent artists at this time who were producing digital artworks, such as Fiona Banner and Julian Opie. The software runs on a custom-made PC built by the company Torch Computers Ltd, the case of which has been professionally resprayed. The hardware used includes a VIA Ezra 800 Mhz processor, 126 MB of RAM and an Intel 845 graphics chip.

The images that appear in the work are sourced from digital versions of Craig-Martin's signature line drawings in the Adobe Illustrator Artwork format. These were imported to Macromedia Director 8 as vector graphics, and can be individually animated using code, which determines the appearance and disappearance of the images. This code was written in Lingo, the high-level scripting language native to the Macromedia Director 8 software. Source code analysis reveals that the parameters of the software behaviour are complex and place limits on the randomisation on the fading of the objects. For example, the number of objects visible at any one time and the speed and regularity with which they appear and disappear are all managed by the code.

Becoming was the first software-based artwork to enter Tate's collection and as such presented a host of new conservation challenges. In 2010, with an interest in assessing the suitability of migrating software to another technology in order to slow the effects of obsolescence, Tate worked closely with the artist and AVCO to

develop contemporary software that maintained the behaviour and formal characteristics of the original. The software was rebuilt in Adobe Flash Professional CS5.5, with the code reimplemented in the ActionScript 3 scripting language with the use of a third-party extension library called GreenSock. The computer case uses a very similar design to the original, with hardware upgraded to an Intel Celeron 1.8 Ghz processor and 248 MB RAM.

This new version of the software addressed several conservation concerns. Since the original software was developed, use of Shockwave, Director and associated technologies had declined in favour of Flash. Furthermore, the timings of the animations in the Shockwave version were dependent on the CPU speed of the host computer, which resulted in problems replicating the intended animation speed on modern hardware. Absolute timings were implemented when the code was re-written in ActionScript. Alongside rigorous documentation of the work, the conservation team has tested a number of other conservation strategies, including virtualisation and emulation.

11.3. Cory Arcangel - Colors (2005)

Colors is a software program which processes a video file—the 1988 film of the same name—transforming it into bands of animated color which are projected in the exhibition space. The software program itself is a Mach-O application for Apple's Mac OS X operating system and utilises the QuickTime and OpenGL visual frameworks which are part of this platform. QuickTime is used to decode the video file and store the current pixel line (starting in the middle of the first frame) in a buffer, which is then projected as a texture map using OpenGL and stretched to fill the screen. The video file itself is a QuickTime MOV format DVD transfer of the film *Colors* by Dennis Hopper, encoded in H.264 video with PCM audio. The audio of the video file is decoded and played back by the QuickTime framework as normal.

The software is written in the C++ and Objective-C programming languages. The artist developed the software using a template from the open-source OpenFrameworks toolkit for the Xcode integrated development environment (IDE). The source code, which was also acquired by Tate, contains code comments, including the comments present in the original OpenFrameworks template and portion of comments made by the artist. Arcangel has clearly marked these, evidently intending the code to read, using the format "<CORY>" to open and "</CORY>" to close these sections, in a playful reference to the syntax of markup

languages such as HTML.

Arcangel has stated that he considers this work the concept of playing back the video file line by line, stretching that line to fill the projection area, and doing this until each line has been played (Arcangel, 2012, March 14). Should it become impossible to run the software on contemporary platforms, this theoretically permits the rewriting of the software in another programming language. This understanding of material significance meshes with our understanding of Arcangel's practice. Arcangel has talked about his artworks as DIY recipes (Birnbaum, & Arcangel, 2009) and has expressed an affinity with open source culture—much of his artwork source code is available online (Arcangel, 2013) and in printed publications (Arcangel, 2017).

Colors is closely related to another artwork by Arcangel called *Colors: Personal Edition*, which has been distributed online as free and open-source software. This work differs conceptually in that the user play back any appropriately encoded video file using the software. Source code analysis indicates that this version differs only in one line, which bypasses the black letterboxing found in the source DVD transfer of the *Colors* movie. The binaries and source code for *Colors: Personal Edition* are available online (Arcangel, 2017), where it is also presented within a corporately styled website aping software culture of the time (Arcangel, 2009).

The work is displayed at a 16:9 aspect ratio and size of at least 14 feet across, in a darkened exhibition space. Stereo speakers are mounted on the walls on either side of the projected image. Since it was last displayed, the QuickTime and OpenGL framework have both been deprecated in newer versions of MacOS, and it is expected that at some point support will be completely dropped by Apple. This means that at some point in the future, should appropriate emulation options not become available, the work may need to migrated to a new software implementation in order to keep it running.

11.4. Rafael Lozano-Hemmer - Subtitled Public (2005)

Subtitled Public is a complex interactive installation, consisting of numerous components brought together in a physical exhibition space. At the heart of the work is custom software running on a network of what the artist calls "surveillance pods", each of which consists of a computer connected to a surveillance camera and projector—the precise models of which are to some degree flexible.

The hardware set acquired with the work consists of a set of mid-2007 Mac Mini (Macmini2,1) shuttle computers running Windows XP (via Mac OSX Bootcamp), a set of Firewire 400 Allied Vision Guppy F-033C surveillance cameras and associated wide angle lenses, and a set of compact, short throw Canon LV-7265 projectors. These computers are networked via an unmanaged D-Link DGS-2205 ethernet switch, using CAT-5e ethernet cables, to a master computer. Each pod can cover a certain maximum area depending on the hardware used (such as camera field of view), which must be considered when installing the work to ensure that zones of surveillance are appropriately configured.

The custom software programs used consists of three 32-bit Windows Portable Executables. The first of these, the “Master” program runs from the central master computer, and manages and controls the networked pod computers, the layout of the space and the assignment of subtitle words. The other networked computers run the “Slave” program, which finds targets for tracking within the camera feed and relays that information back to the master computer over the network. Finally, a separate camera calibration program is used in the installation to correctly configure the camera's position and orientation and correct for radial distortion.

The two software programs running on the pod computers use the Microsoft DirectShow interface, which is a part of Microsoft’s DirectX framework, to access the camera feeds. Each computer is assigned a static local IPv4 address, a software communication is carried out using the UDP protocol. A considerable amount of configuration can be undertaken once the software is installed, allowing flexibility in the way the software is installed. The word list defined is not an exhaustive list of conjugated verbs and does not include unusual or particularly complex verbs. The software uses the Arial font at a dynamically sized scale for the formatting of the words.

The custom software was developed in the Borland Delphi 7 integrated development environment (IDE) and coded in the derivative of the Object Pascal programming language that this environment supports by Conroy Badger, a programmer who has collaborated with the artist on a number of projects. In its development a number of open-source computer vision libraries were employed, including Intel’s Open Source Computer Vision (OpenCV) Library and Image Processing Library. Badger has stated that Delphi and the UDP protocol were

chosen due to their reliability for real-time applications such as the time-sensitive tracking carried out in Subtitled Public (Badger, 2008). The DirectShow component of the code was based on the Amcap program (original in the language C), while the UDP component uses the open source Indy library for Delphi.

The artwork can be installed in a variable exhibition space, and the number of surveillance pods adjusted to meet different size requirements. The space is dark, but uses illuminators fitted with congo-blue filters to provide a low blue light, which improves the visibility of targets on the infra-red sensitive cameras. The tracking system is sensitive to cast shadows, and so requires careful management of lighting sources and wall and floor surfaces when installed.

The artist has stated that while he is very happy with the implementation described above, it is the concept of “subtitling the public” which is his primary impetus behind the creation of the work, and as such it is not linked to a specific implementation of the software or a particular set of hardware (Lozano-Hemmer, 2006). As a result, there is a certain amount of room for altering components of the system in order to cope with obsolescence in the future. The artist has also stated that, with consultation, he would welcome improvements to latency, stability and precision of tracking.

In 2018, the work was migrated to a new set of hardware, consisting of a set of Intel i3 NUC PCs running Windows 10, IDS Imaging uEye LE USB 3.1 surveillance cameras and BenQ MP771 Projectors. The original Delphi software was used, demonstrating that for now it is possible to run the it in contemporary technical environments despite the time elapsed since the work was authored. For how long is unclear, however, as DirectShow, the means of accessing the cameras, has been deprecated by Microsoft and may be dropped from future versions of Windows.

11.5. Jose Carlos Martinat Mendoza - Stereo Reality Environment 3: Brutalism (2007)

The primary formal focus of *Brutalism* is the wooden scale model of the Pentagonito building. This is a free-standing structure made up of 12 box like elements, each of which is constructed from glued and screwed pieces of MDF. A variable number of Nanoptix High-Speed Kiosk thermal printers (typically used for till receipt printing) are placed on the top of this sculptural component. These printers are connected to a repurposed Dell Workstation PC (by a visible mass of cables), running the Ubuntu 7.04 operating system (with Gnome 2.18.1 desktop), which sits on a floor next to the

model. This computer is connected to the internet, and runs software which gathers internet search results relating to the term “brutalism”, and prints fragments of these onto slips of paper which fall to the gallery floor and accumulate during the works display.

The software involves two custom Java applications, each of which carries out a particular function. The first is the internet search harvester component, which uses the Google Search API to make queries based on pairings of words. The first is "brutalism", which is then combined with a second word randomly selected from a set of 11 related terms (such as “Concrete”, “Blood” and “Torture”). Text is harvested as HTML from the first sentence to contain the words within the page results, stripped of HTML tags by a special parser component (HTML Parser 1.6), and then stored in a SQL database managed by MySQL 5.0.38.

The second Java application manages the retrieval of terms from the database, and communication with the printer. Using SQL queries sent via the JDBC API, the application retrieves random text fragments from the database, and sends them to a random printer using the parallel port interface and DB-25 connectors. Printing is able to continue independently of the internet search component. The same database is used each time the work is installed, and so allowing it to grow. When installed, sensors are sometimes used to limit the regularity of printing to only occur when gallery visitors enter the space.

The software was developed in the NetBean’s 5.51 integrated development environment (IDE), by the artist and programmer, Arturo Diaz Rosemburg. Using this IDE provides certain benefits to the programmer, as it is designed for working with Java programming projects. The function of the software within Brutalism is similar to that of other works by Martinat, which employ internet searching and printer. In this case, code from an earlier work in the *Estéreo Realidad* series called *Inkarri* was used as a basis on which to build, artefacts of which are present in some of the Java class and module names.

Using remote access tools to work on the computers at Tate, modifications have been made to the software at various points in time, particularly in the run up to its installation at Tate Modern in 2011. This resulted in the implementation of USB printer support, to ensure support for printers which do not use the now obsolete parallel port interface. Modifications have also been made to keep up to date with changes to the Google Search API. As such, this work must continue to evolve at

the software level, in order to remain functional in a changing technical environment.

11.6. John Gerrard - Sow Farm (2009)

Sow Farm employs a medium the artist calls real-time 3D. This involves the use of a system of computer hardware and software to render a 3D environment in which events unfold in real-time. The custom software at the heart of the system is a 32-bit Windows Portable Executable file, associated with a set of text files which allow manual configuration of certain elements of the simulation and rendering. The executable file encapsulates the data assets (such as 3D models and textures) which are used to realise the 3D environment, as well as the proprietary rendering engine and the simulation model which controls the day-night cycles. The software was developed for Windows 7, and requires access to additional supporting software on the host system including the Phidget21 libraries, DirectX 9 helper libraries and Microsoft Visual C/C++ runtime libraries.

Gerrard worked with a production team at his studio in Vienna to create Sow Farm. The development of the software involved a team including a production lead (Werner Poetzelberger), 3D modeller (Daniel Fellsner) and programmer (Helmut Bressler). An engine and authoring tool called Quest3D (in this case version 3.6.6) was used to create the software. Quest3D provided a development environment for the creation of 3D software, through the simplification of some of the more complex aspects of working with 3D graphics. This engine would have been typically used for purposes such as architectural visualisation and video game development.

In addition to the engine at the heart of the development process, a number of other processes and tools were utilised in the multi-stage production process. As for other works of this kind by Gerrard, this began with research photography undertaken in the field at a real-world pig farm. Using this material, 3D assets were created in Maya and 3D Studio Max, two industry standard software tools for 3D modelling and animation. These assets could then be imported into Quest3D as DirectX .X files. Textures were created in Adobe Photoshop and imported into Quest3D as Direct Draw Surface (DDS) files. Surfaces in the environment consist of a number of texture layers, including diffuse (colour), specular (colour and intensity), normal (light mapping) and transparency (alpha). Ambient occlusion information, used to help achieve realistic shadowing, was baked into the diffuse texture layer. The models and textures were assembled as a scene in Quest3D, where lighting and custom shaders–a way of implementing 3D rendering effects–written in the HLSL

programming language were added. For example, the grass effects are achieved using a repurposed shader for generating animal fur.

The work is usually projected at a resolution of 1600x1200 pixels (4:3 aspect ratio) when installed, in a light locked room with flooring that reflects some of the projected image. The computer hardware used to run the software is flexible between installations, although Gerrard has specified that it should be able to maintain an output frame rate of at least 60 frames-per-second at all times. When installed in 2015 at Tate Britain, a PC running Windows 7 Professional (64-bit) was used with an Intel Core i7 4820k processor, 16GB RAM and an NVIDIA GTX 780 graphics card. The NVIDIA graphics card driver was used to apply a number of graphical effects, specified by Gerrard, to the rendered output. These included multi-sample anti-aliasing (reducing edge artefacts), anisotropic filtering (improving texture detail at angles) and vertical sync (locking frame rate to refresh rate). The software uses the host machines system clock on which to base the time of day simulated in the 3D environment, which is set to the real-time in Oklahoma when the work is on display. The appearance of the truck operates on a separate time scale, triggering after the software has been running for 159 days without interruption.

Since creating *Sow Farm*, Gerrard's production process has continued to evolve and Quest3D is no longer used to create new artworks. Furthermore, the support and sale of Quest3D has been discontinued by its owner, Act-3D. This means that it is likely that this proprietary software will no longer be updated to function in contemporary hardware and software environments, increasing risk of obsolescence. Conservation research at Tate has explored the use of virtualisation as a means of preserving the work. Virtualisation enables suitable hardware to be simulated, thus allowing the long-term operation of an appropriate technical environment in which to run the software.

APPENDIX IV: LITERATURE SEARCH STRATEGIES AND TERMS

12.1. Literature Search Strategy

Literature was reviewed in a series phases, the first of which focused on an initial shortlist of search categories identified in collaboration with the project supervisors. This phase was focused on identifying relevant research within the high-level categories of art conservation and digital preservation, and the state of the art in the conservation of software-based art, which operates at their intersection. Further phases of literature review were carried out relation to specific areas of interest identified in later chapters, which served to fill gaps identified in existing scholarship relating to the conservation of software-based art. The high-level search categories identified were media theory (incorporated in Chapter 2), information science (incorporated in Chapter 3) and software engineering (incorporated throughout Chapters 4, 5 and 6). Further information on search terms and their formulation is detailed in Section 12.2.

Searches were primarily undertaken using academic search engines, and to a lesser extent the digital and physical library indexes available at King's College London Library, Tate Library & Archive and Senate House Library. Google Scholar was the main academic search engine employed, as in initial tests it was found to return

results equivalent or superior in quantity when compared to other options tested during literature review. The multidisciplinary or humanities-specialist indexes Base, CiteSeerX, Project MUSE, Scopus, and Web of Science were all tested. For computer science and software engineering related topics the IEEE Xplore Digital Library was also used. Due to the close link between this research and a field of practice of which a considerable amount of research existing in non peer-reviewed publications, searches were also passed through Google's general search engine to ensure these important sources were not missed. Boolean operators were used frequently in all searches to narrow down results, as were conjunctions of terms and experimentation with alternative phrasings. Given their frequent conflation, for cases where the terms "conservation" or "preservation" were used, search variations using both terms were carried out.

Literature identified was managed using the Zotero reference manager platform, where it was grouped into libraries based on the search categories. Zotero and plugins for LibreOffice Writer and Microsoft Word were used for managing references and compiling the final bibliography.

12.2. Table of Search Categories and Terms

Term Category	Term	Sub-term
Art Conservation	Time-based Media Art Conservation	Software-based Art Conservation
		New Media Art Conservation
		Internet Art Conservation
	Managing Change in Art Conservation	
	Authenticity in Art Conservation	
	Technical Art History	
	Art Conservation Documentation	Installation Documentation
		Acquisition Documentation
Digital Preservation	Significant Properties	Significant Properties of Software
		Significant Knowledge
	Software Preservation	
	Digital Preservation Documentation	Digital Preservation Metadata
	Digital Preservation Strategies	Emulation in Digital Preservation
		Migration in Digital Preservation
		Storage in Digital Preservation
Media Theory	Software Studies	

	Digital Forensics	
	Media Art History	Computer Art History
		Digital Art History
		Software Art History
	Digital Materiality	
Information Science	Modelling	Lifecycle Modelling
		Continuum Theory
		Software Modelling / Computer Systems Modelling
	Documentation Theory	Documentation and Representation
		Documentation Science
		Digital Documentation Theory
	Museum Documentation	Cataloguing
		Information Systems
	Structured Documentation	Metadata
		Vocabularies
		Ontologies
Software Engineering	Requirements Engineering	Functional Requirements
		Non-functional Requirements
	Software Evolution	Software Versioning
	Software Analysis	Dynamic Analysis / Process Analysis
		Static Analysis / Binary Analysis
	Software Reverse Engineering	Decompilation
		Source Code Analysis
	Legacy Software	
	Program Comprehension	

Table 9. List of primary search terms employed in the literature review undertaken during this research.